TASK: UT40
CDRL: 4041
22 Nov 91

UT40 — Asset Library Open Architecture Framework (ALOAF) Version 0.8 - *DRAFT*

Informal Technical Data

STARS-TC-04041/001/00

22 November 1991

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE 22 November 1991	3. REPORT TYPE AND DATES COVERED Informal Technical Data

4. TITLE AND SUBTITLE Asset Library Open Architecture Framework ALOAF Version 0.8	5. FUNDING NUMBERS F19628-88-D-0031
6. AUTHOR(S) Paramax Corporation	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Paramax Corporation 12010 Sunrise Valley Drive Reston, VA 22091	8. PERFORMING ORGANIZATION REPORT NUMBER STARS-TC-04041/001/00
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Department of the Air Force Headquarters, Electronic Systems Division	10. SPONSORING/MONITORING AGENCY REPORT NUMBER 04041
11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION/AVAILABILITY STATEMENT Distribution "A" **This document has been approved for public release and sale; its distribution is unlimited.**	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The STARS (Software Technology for Adaptable, Reliable Systems) Asset Library Open Architecture Framework (ALOAF) addresses the exchange of reusable assets among diverse libraries, and the definition of an asset library platform upon which portable reuse tools may be constructed. Asset interchange and asset service interfaces are critical elements in achieving a broader objective—asset libraries which interoperate to such an extent that the boundaries between individual libraries become invisible to the end user. In general terms, this is the STARS vision of "seamless" library interoperation. As the STARS vision of seamless interoperation matures, the ALOAF will be updated and expanded to address additional, more advanced requirements.

14. SUBJECT TERMS ALOAF	15. NUMBER OF PAGES 76
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT unlcassified	18. SECURITY CLASSIFICATION OF THIS PAGE unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT unclassified	20. LIMITATION OF ABSTRACT SAR

TASK: UT40
CDRL: 04041
22 November 91

INFORMAL TECHNICAL REPORT

For The

SOFTWARE TECHNOLOGY FOR ADAPTABLE, RELIABLE SYSTEMS (STARS)

Asset Library Open Architecture Framework
Version 0.8 - **DRAFT**

STARS-TC-04041/001/00
Publication No. GR-7670-1317(NP)
22 November 91

Data Type: A005, Informal Technical Data

CONTRACT NO. F19628-88-D-0031
Delivery Order 0008

Prepared for:

Electronic Systems Division
Air Force Systems Command, USAF
Hanscom AFB, MA 01731-5000

Prepared by:

The Boeing Company,
Defense & Space Group,
P.O. Box 3999, MS 87-37
Seattle, WA 98124-2499

IBM,
Federal Sector Division,
800 N. Frederick Pike,
Gaithersburg, MD 20879

Unisys Defense Systems, Inc.,
Tactical Systems Division,
12010 Sunrise Valley Drive,
Reston, VA 22091

TASK: UT40
CDRL: 04041
22 November 91

Data ID: STARS-TC-04041/001/00

TASK: UT40
CDRL: 04041
22 November 91

INFORMAL TECHNICAL REPORT
Asset Library Open Architecture Framework
Version 0.8 - **DRAFT**

Approvals:

Margaret J. Davis

Boeing Reuse Technical Lead *Margaret Davis* *Date*

[illegible]

IBM Reuse Technical Lead *Brian Bulat* *Date*

[illegible]

Unisys Reuse Technical Lead *Richard Creps* *Date*

(Signatures on File)

TASK: UT40
CDRL: 04041
22 November 91

INFORMAL TECHNICAL REPORT
Asset Library Open Architecture Framework
Version 0.8 - **DRAFT**

Change Record:

Data ID	*Description of Change*	*Date*	*Approval*
STARS-TC-04041/001/01	Updated: Specifies additional services.	22 November 1991	*on file*
STARS-SC-03727/001/01	Reissued: Electronic representation enhanced for improved printability; corrected minor formatting discrepancies.	05 September 1991	*on file*
STARS-SC-03727/001/00	Original Issue	21 August 1991	*on file*

Contents

List of Figures

Prologue

This is the second draft, version 0.8, of the Asset Library Open Architecture Framework (ALOAF). This document supersedes the first draft, version 0.5, released in August 1991. Version 1.0 of the ALOAF will be available in February 1992.

This second draft version includes a specification of the asset library meta-model that underlies the framework services. It also contains a significantly expanded and reorganized framework services section. The ALOAF document is also being re-examined in light of the recent NIST and ECMA work on reference models for CASE frameworks. Revisions for version 1.0 are expected to reflect this work.

The ALOAF is a "living" document. The document will be revised as necessary to reflect the current status of and consensus about open architecture frameworks for asset libraries. These revisions may include, but not be limited to, enhancement of the current concepts, addition of new material, deletion of existing material, and even contradiction of premises stated in previous versions. Knowledge gathered through prototyping efforts and experiments on the gamut of issues associated with reuse, along with feedback from internal and external review will serve as a basis for modification and extension of this document.

Part I

The ALOAF Concept

1 Introduction

A reuse-based approach to software engineering places the emphasis on the reuse and integration of existing software components and systems, rather than the creation of software components from scratch. To support this approach, automated reuse libraries have been and are being created.

The concept of reuse of components is applicable to reuse libraries themselves. Reuse libraries consist of a set of components that are suitable candidates for reuse and sharing. These include the components that make up an automated library system and reuse library tools.

The STARS (Software Technology for Adaptable, Reliable Systems) Asset Library Open Architecture Framework (ALOAF) addresses the exchange of reusable assets among diverse libraries, and the definition of an asset library platform upon which portable reuse tools may be constructed. Asset interchange and asset service interfaces are critical elements in achieving a broader objective—asset libraries which interoperate to such an extent that the boundaries between individual libraries become invisible to the end user. In general terms, this is the STARS vision of "seamless" library interoperation. As the STARS vision of seamless interoperation matures, the ALOAF will be updated and expanded to address additional, more advanced requirements.

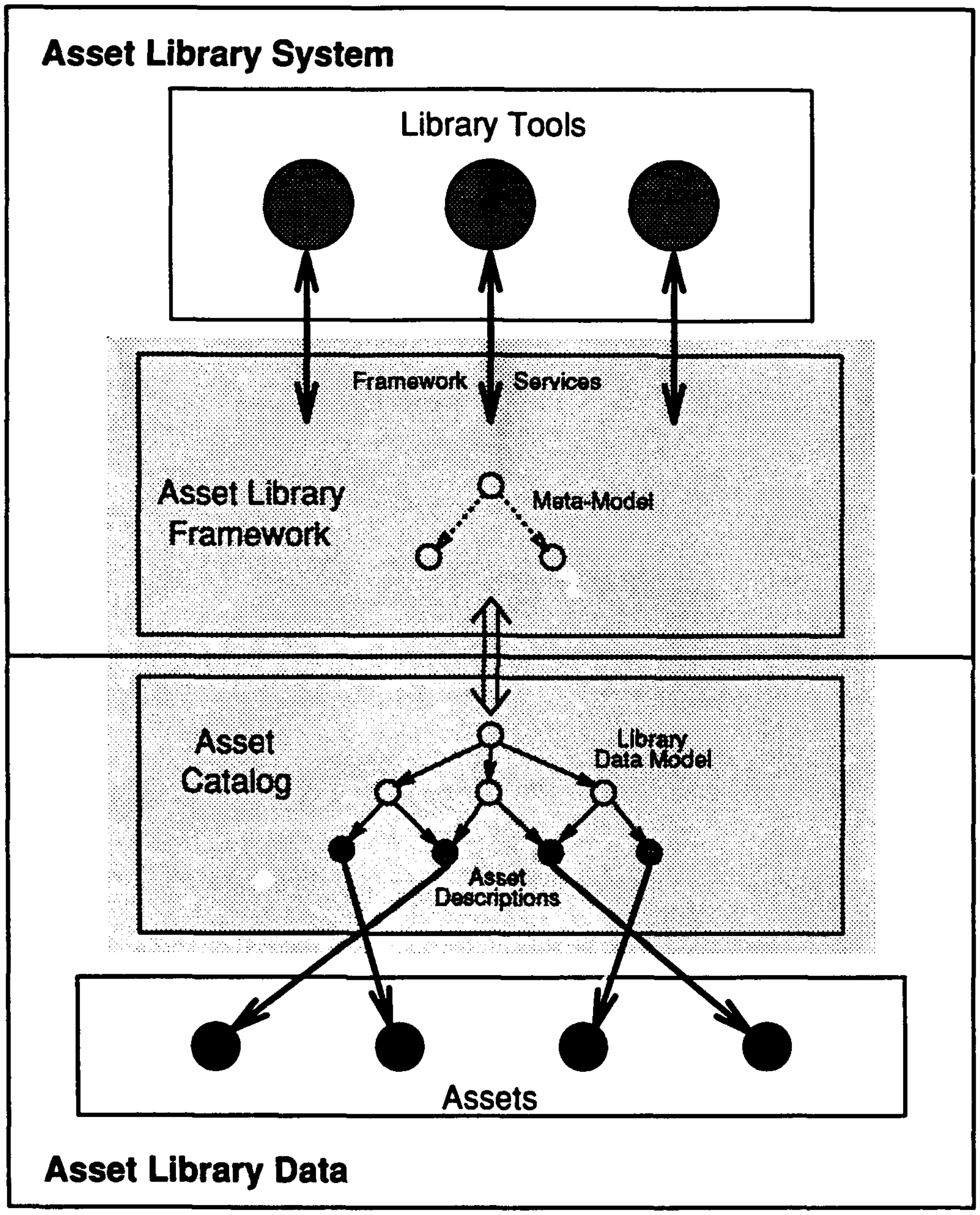
1.1 Asset Library Overview

An *Asset Library*, as depicted in Figure 1, is a means for organizing, collecting, and managing a set of engineering components, or *Assets.* An *Asset Library* consists of the *Asset Library Data* and the *Asset Library System.*

The *Asset Library Data* consists of the *Assets* themselves, as well as the descriptive and organizational information about the *Assets*, collectively referred to as the *Asset Catalog* in Figure 1. The *Asset Catalog*, in turn, contains the descriptive information (*Asset Descriptions*) associated with the *Assets*, and the structural organization (*Library Data Model*) of the *Asset Descriptions.*

The *Asset Library System* provides a set of library-oriented mechanisms which define, manipulate, and operate upon the *Asset Library Data.* The *Asset Library Framework* and *Library Tools* make up the *Asset Library System.* The *Asset Library Framework* provides the basic library-oriented operations, the *Framework Services*, that are needed to create, manipulate, and manage the *Asset Catalog*, in accordance with a data organizational structure defined by the *Meta-Model.* The *Library Tools* provide the end-user or other tools a higher level, organized collection of operations. These tools may operate on the *Asset Catalog* and/or the *Assets.*

The area of focus of the ALOAF includes the *Asset Library Framework* and the *Asset Catalog.* An asset library open architecture framework specifies a public, non-proprietary blueprint, against which asset library systems may be implemented and reuse-based library tools may constructed.



▨ = Area of ALOAF focus

Figure 1: Asset Library

1.2 Objectives

The primary objectives of the STARS ALOAF are to facilitate the interchange of assets between asset libraries, and facilitate the construction of reuse tools that are portable between asset libraries. The asset-interchange objective focuses the STARS ALOAF upon the information needed to systematically organize and describe assets stored within an asset library. The ALOAF addresses the interchange of assets and their associated asset descriptions and model information through the ALOAF Data Modeling and Asset Interchange Specification. The portable-reuse-tools objective focuses the STARS ALOAF upon the asset library services and standard interfaces needed by reuse-based library tools. The ability to create portable reuse-based library tools is addressed by the ALOAF Service Model along with their ALOAF Programmatic Interfaces.

ALOAF Data Modeling Concepts and Asset Interchange Specification

An asset library can contain a large amount of data. This data may include the library's assets, descriptions or related information about the assets, as well as the organization of the assets and the manner in which the organization is described. The ALOAF Data Modeling Concepts address all of these constituent pieces of data via three layers: a data layer, a model layer, and the meta-model layer. Data modeling is necessary in order to specify a common asset interchange mechanism, as well as a uniform set of services (the ALOAF Service Model) which operates upon the data.

The ALOAF Asset Interchange Specification supports the interchange of assets and asset descriptions among diverse asset libraries. The emphases within asset interchange are upon the exchange of the asset descriptions and their organizational representation, and upon an open, non-restrictive interchange mechanism.

The STARS ALOAF Asset Interchange Specification provides a standard technique for representing library data models, a format for library data models, and a format for asset library data. The Asset Interchange Specification is not dependent upon any particular library data model and may be used to represent the data models and data of a wide range of asset libraries. The Asset Interchange Specification includes a Common Data Model. The Common Data Model describes a basic data model that allows asset libraries to interchange a common subset of asset descriptions. The Common Data Model encompasses information that is typically maintained by asset libraries.

ALOAF Service Model

The ALOAF Service Model describes a collection of services that asset library implementors are encouraged to provide to support interoperability among geographically dispersed, heterogeneous asset libraries and to support portability of tools across libraries. The Service Model categorizes framework services and describes the interrelationship between categories and between individual services within and across category boundaries. Individual services are described in terms of service protocol descriptions that are independent of implementation language. The service protocols consist of abstract functional interfaces and data exchange specifications intended to meet requirements common to all language bindings to the ALOAF services. STARS asset libraries will be required to conform to the ALOAF. Conformance descriptions and criteria for asset libraries will be specified as an extension of the ALOAF Service Model.

ALOAF Programmatic Interface

The ALOAF Programmatic Interface comprises a set of Ada package specifications defining interfaces to the services described in the ALOAF Service Model. In particular, the Ada specifications provide an Ada instantiation of the implementation-language-independent service protocol descriptions articulated in the Service Model. The Ada programmatic interface is only one of many potential language-specific instantiations of the service protocols and is provided by STARS as a recommended Ada standard interface. However, ALOAF conformance is not predicated on provision of ALOAF services through Ada interfaces.

1.3 Evolution

STARS has invested substantial effort in the incorporation of reuse methods and technologies into the mainstream of software engineering. Projects concentrating on the advancement of software reuse have demonstrated successful results over the past five years in the STARS Foundations, STARS Prime, and other STARS programs.

The concept formulation of an ALOAF started in the fall of 1990, and came to realization several months later. The ALOAF effort began in earnest in January 1991, with participation from all three STARS Prime participants—Boeing, IBM, and Unisys. This ALOAF is the result of the cooperation and confluence of reuse efforts of the three Primes, their subcontractors, and other STARS program participants.

The initial ALOAF addresses the reuse goals of STARS asset libraries with both short term and long term solutions. The short term solutions drive toward the immediate sharing of assets between the many diverse reuse libraries that are coming on-line in increasing numbers. The long term solutions strive for the far reaching goals of interoperable heterogeneous asset libraries, able to share assets and asset descriptions, as well as the portability of a broad and rich set of library tools which operate within asset library systems.

The ALOAF document will track the development and expansion of reuse within the software community. The framework to support asset interchange and tool portability will be revised as necessary to reflect the state of the practice, while tracking advances in the state of the art.

1.4 Document Outline

The remainder of this document describes the STARS ALOAF. Sections 2 through 4 of Part I, The ALOAF Concept, briefly describe the background of the STARS ALOAF, the assumptions upon which the STARS ALOAF is based, and the high level aims of the STARS ALOAF.

Part II, The ALOAF Library Model, is composed of Sections 5 and 6. Section 5 introduces the data modeling concepts of the ALOAF and Section 6 describes the ALOAF service model.

Part III, The ALOAF Specifications, is composed of Sections 7 through 10. The ALOAF data modeling along with the asset interchange specifications are described in Section 7. Section 8 provides a detailed description of the set of ALOAF services, followed by a discussion of criteria for conformance with the ALOAF in Section 9. Section 10 concludes the main body of the ALOAF with the references for this document.

Part IV consists of the appendices. Appendix A is a glossary of ALOAF related terms, and Appendix B defines the acronyms frequently used within this document. Appendix C provides a sample scenario(s) which illustrates the usage of the ALOAF services and ALOAF asset interchange. Appendix D defines the asset interchange language, and Appendix E specifies the Ada binding to the language independent specifications of the ALOAF services.

2 Background

The ability to make effective use of previously created assets, be they software components, design artifacts, textual documents, etc., is viewed as a critical factor in the reduction of application development and maintenance costs, along with improved reliability. Many efforts both within and external to the STARS program have addressed the development and establishment of asset libraries (also referred to as reuse repositories, software depositories, ...). Each of these reuse projects has its respective merits and unique qualities. The ALOAF does not seek to stifle the creativity nor inventiveness being exhibited in the development of new reuse methods, environments, and tools. Rather, the purpose of the ALOAF is to allow reuse projects to benefit cooperatively from each other's work. Examples of such cooperative benefit are:

> Assets stored within one asset library may be interchanged with a completely different asset library, with descriptive asset information being exchanged as well. This allows diverse, heterogeneous asset libraries to share assets, enabling the construction of applications which may make use of the best and newest components that are available.
>
> A reuse tool created for one asset library system may be easily ported to another asset library system when the reuse tool and the asset library systems conform to the ALOAF service model interface. This allows asset library systems to be easily enhanced with additional reuse-based tools.

Thus, reuse technology, methods, and assets as a whole may rapidly expand and facilitate a shift to reuse-based development and engineering.

2.1 STARS Reuse

Reuse, along with software engineering environments and processes, is one of the primary software technical areas being addressed within the STARS program. The vision of the STARS program [STA91b] is that "Software-intensive system development will evolve to a process-driven, domain-specific reuse-based, technology-supported paradigm." The element of the STARS vision that explicitly applies to the ALOAF is *domain-specific reuse-based.* Under this envisioned paradigm, applications will be built using a component based approach, rather than constructing them a line at a time. Reusable components along with components extracted from existing systems will be assembled into an application. And the components used in the application construction will be based on domain-specific architectures and open interface standards.

In order to support DoD needs, a key STARS technology objective is the construction of engineering environments from commercially available environment frameworks and tools. Asset library systems contain elements of both reuse frameworks and reuse tools. The desire within STARS is the construction of modular reuse tools and frameworks that conform with an open architecture, hence the formulation of the asset library *open architecture* framework. Environment assemblers may then pick and choose from a variety of commercial-off-the-shelf or company-proprietary tools and systems that conform to the STARS ALOAF, with the knowledge that all of these ALOAF-conformant components will interoperate and be plug-compatible. The resultant engineering environments are standards-driven and standards-based systems, which are tailorable, adaptable, and reliable.

The ALOAF focuses on the needs of STARS asset libraries, as well as other DoD-related asset/reuse systems with which STARS asset libraries will interoperate. This document is based upon the past, current, and near-term anticipated work related to the asset library systems of the STARS program. It was beyond the scope of the initial work of the ALOAF to address and consider all reuse issues associated with every existing and potential asset/reuse library. In future work, the scope of the ALOAF will be broadened to encompass additional relevant and appropriate asset/reuse libraries, based upon the levels of interest demonstrated by those associated with those reuse libraries.

Other STARS, reuse-related activities include the STARS Reuse Concept of Operations and STARS' participation in the Reuse Library Interoperability Group (RIG). The STARS Reuse Concept of Operations [STA91a] articulates STARS concepts and expectations with respect to reuse of software related assets across system and software life cycles. Specifically, the document communicates the joint STARS perspective on such topics as the reuse vision, the goals for reuse, and reuse processes. The ALOAF is consistent with and fosters the realization of these joint STARS perspectives put forth within the Concept of Operations. The purpose of the RIG is to facilitate the interoperability of government-sponsored software reuse libraries [RIG91]. As STARS is a member of the RIG, relevant portions of the ALOAF will be put forward as suitable candidates for consensual standardization within the RIG.

2.2 Relationship to SEE

An asset library system is one of many subsystems that may be part of a computer-aided software engineering (CASE) environment. As a subsystem within a CASE environment, the facilities provided by an asset library system should be consistent with the reference model defined for CASE environment frameworks. The ECMA (European Computer Manufacturers Association) has defined a reference model for CASE environment frameworks [Ear90], and the ALOAF has adapted that model as the basis for part of its specification. The ALOAF differs from the ECMA reference model for CASE environment frameworks in three important ways:

a. The ALOAF addresses asset library frameworks rather than the broader scope of CASE environment frameworks.

b. The ALOAF addresses both asset library framework services and asset library data.

c. The ALOAF contains not only a model for asset library services and data, but also specific STARS specifications for providing those services and describing that data.

The ALOAF supplements a Software Engineering Environment (SEE) in the area of asset libraries and reuse services. The ALOAF service model defines the basic reuse capabilities needed within asset library systems. Many of these reuse functions are intrinsic to asset libraries and in some cases are strongly interrelated with object management services. The ALOAF services are a layer above the SEE services when viewed within a SEE context, thus providing value-added, reuse-specific capabilities. The STARS ALOAF has been constructed such that asset library systems may utilize those facilities that are provided by a software engineering environment. These facilities include, but are not limited to, a user interface system, an object management system, and data transport services.

A STARS SEE is a possible underlying substrate upon which an ALOAF-compliant asset library system may be constructed. Within the STARS program, there is a high degree of interest in the construction of just such an asset library system. However, the ALOAF's broader goal is to address asset library frameworks for systems that are outside of STARS as well. In order to make the ALOAF a reasonable candidate for standardization within the broader reuse community, the needs of library developers outside of STARS must be taken into account. Not all library developers may have STARS SEEs at their disposal, but may wish to interoperate with other libraries and reuse tools. Thus, the existence of a STARS SEE is not a necessary condition within the ALOAF.

2.3 Standards Activities

One aspect of an open system is the adherence to and conformance with standards relevant to the technical application domain. A goal of the Asset Library *Open* Architecture Framework is the adoption of existing and/or emerging standards that are relevant to asset library systems. Standards activities that are relevant to the goals of the ALOAF are currently being tracked and analyzed. Those standards activities that have direct bearing or possible secondary effects on the goals of the ALOAF will be seriously considered for adoption and incorporation into the ALOAF. It is not the intent of this document to duplicate or reinvent work that has already been completed or is in progress.

The specification and promulgation of new and emerging standards is just as important as, if not more important than, the conformance to standards. Reuse technology is one area in which there are relatively few existing standards. A primary purpose of the ALOAF is the development and shaping of future reuse-related standards. A role of the ALOAF is to serve as initiator and catalyst in building and shaping consensus on reuse technology standards among asset library developers.

3 Assumptions

This section identifies various kinds of assumptions that have been made in establishing the requirements for the ALOAF and in the development of the ALOAF. We expect that these core assumptions will increase in number and change as we discover the unstated assumptions that have been made, and as the level of detail of the ALOAF increases. This section is organized by topics about which assumptions have been made.

3.1 DoD Needs

The DoD will require reuse-based software development and maintenance to achieve the productivity and quality that are needed in the 1990s. DoD programs will involve multiple, geographically distributed organizations using heterogeneous software engineering environments. Reusable assets must be shared among these organizations.

3.2 Asset Library Technology Standardization

The asset library technology to support DoD needs is immature. Several reuse library mechanisms have been independently developed. There has been little effort thus far to standardize any aspect of reuse libraries or of asset exchange among libraries.

3.3 Assets

The reusable assets assumed to be in asset libraries include not only the software components most commonly associated with reuse, but also such additional kinds of information as the following:

- Reusable forms of other software products; e.g., requirements specifications, architectures, designs, test procedures
- Application domain knowledge; e.g., models, data dictionaries, algorithms
- Process definitions; e.g., for managing asset libraries, for developing application systems
- Rationale; e.g., for the inclusion of features, services, objects, and/or algorithms in a system; for the selection of one architecture or design over another.

3.4 Asset Library Data

Libraries of reusable assets are described by a library data model. It is assumed that not all libraries will have the same data models. Data models may change over time.

3.5 Asset Library Systems

Asset libraries are managed and accessed by asset library systems. The library system includes a basic set of capabilities to create and manipulate library data models and asset descriptions. The asset library system also includes other capabilities to support the management and use of the library. For example, it might include capabilities that support the presentation of information to a user, the use of assets to compose software systems, the browsing of an asset library, or the exchange of assets among libraries.

3.6 Library User Needs

It is assumed that an asset library user searches for various kinds of reusable assets to support some software system life cycle activity. The user needs access to multiple libraries in order to maximize the amount and effectiveness of reuse. The ideal is for the user to use a single interface to interact with all libraries and be unaware of whether or not an asset comes from a local or remote library and of the particulars of the user interface or of the data model associated with the originating library.

4 Aims

The purpose of this section is to describe and provide rationale for the main aims of this document. It is patterned after and adopts some portions of section 1.4 "Aims of the Reference Model" found in the ECMA Reference Model document [Ear90]. Section 4.1 describes the aims for the library model (Part II) portion of the document. Section 4.2 describes the aims for the specifications portion of the document (Part III).

4.1 Aims for the ALOAF Library Model

4.1.1 Description and Comparison

Aim: The ALOAF library model will be suitable for use in describing, comparing and contrasting existing and proposed asset library frameworks or systems.

Rationale: The area of asset library frameworks/asset library systems is immature. Asset library frameworks do not exist today; some asset library systems do exist. The area is just beginning to develop basic premises and terminology. Much effort could be wasted through misunderstandings and misinterpretations when different groups meet to discuss asset library problems or proposals. A major contribution of the ALOAF can be provision of a vehicle to facilitate effective communication about asset library frameworks and systems.

4.1.2 Evolution of Standards

Aim: The ALOAF will provide for the smooth and coordinated evolution of future standards, in particular to ensure that the early standards are developed in such a way that further standards may easily achieve alignment in the sense of upward compatibility.

Rationale: A smooth transition to the widespread use of asset libraries will rely on users being able to continue to use their existing techniques and organizations in conjunction with an asset library framework that will provide more automation and tool support. System developers should be able to see that the standards that they invest in today will not be obsolete tomorrow.

4.1.3 Integration and Interoperability

Aim: The ALOAF will address interoperability and integration of reuse tools.

Rationale: The wide range of tool capabilities required to support various organizations and applications of reusable assets may not be obtainable from a single source. Standards provide a way for asset library users to buy facilities that meet their requirements from various vendors.

Standards will be very useful in enabling tools to be ported to and to interoperate over, different types of hardware.

4.1.4 Degree of Generality

Aim: The ALOAF will be usable to describe a wide range of asset library framework designs, but will also be usable to define points at which useful standards can be defined.

Rationale: The ALOAF library model could be very abstract; but that would make it difficult to define corresponding standards. The model could prescribe a single design; but that would lack generality and defeat the aim of providing a vehicle for communication about various asset library frameworks.

4.1.5 Technique

Aim: The ALOAF will be applicable to asset library systems irrespective of implementation techniques, reuse-based software development techniques, or library data modeling techniques.

Rationale: The ALOAF should recognize that there are many approaches to reuse-based software development and to asset library data modeling.

The ALOAF should not be unduly restricted by existing library systems. There is limited experience in their use to support software development and thus little evidence that their services and designs are the most appropriate. Existing systems were not developed to support seamless access to diverse libraries. However, existing products and results of research projects provide a valuable body of knowledge applicable to the ALOAF.

4.2 Aims for the ALOAF Specifications

This subsection identifies the specific aims for the parts of this document that identify the STARS specifications for asset library interchange and for asset library services (Part III).

4.2.1 Conformance with Standards

Aim: The ALOAF will be consistent with work undertaken by other organizations working toward standardization related to reuse and amenable to adoption by a larger community.

Rationale: The STARS program is providing technology to be used in the development of DoD software systems. The technology can be commercialized, transitioned and used effectively only if it is consistent with community and industry standards.

4.2.2 Asset Interchange

Aim: The ALOAF will create an asset data interchange specification that provides a common form of expression of all information about an asset from heterogeneous libraries.

Rationale: Such a specification is necessary for the STARS asset libraries to exchange asset information in a way that maximizes the usefulness of the exchanged asset.

4.2.3 Common Data Model

Aim: The ALOAF will define a Common Data Model for the expression of asset information common to asset libraries.

Rationale: The existence of a Common Data Model will allow the meaningful automated exchange of some asset information among libraries having different data models. A Common Data Model may also facilitate the implementation of asset library capabilities by providing for some known information about all exchanged assets.

4.2.4 Asset Library Service Categories

Aim: The ALOAF will identify and categorize a set of asset library services that provide basic capabilities to create and manipulate asset library data models and asset descriptions.

Rationale: Identification and categorization of basic asset library capabilities or services will provide a vehicle for communication among parties interested in asset libraries. It will also provide the basis for establishing priorities in the development of specifications.

4.2.5 Language Independent Specification

Aim: The ALOAF will specify services in a language independent form.

Rationale: Specification of the services in a language independent form will ensure that the specifications are not biased towards a single language and that they are understandable to a broad community. Specification in a language independent form is consistent with modern practices.

4.2.6 Ada Programmatic Interfaces

Aim: The ALOAF will specify a set of Ada bindings to the services, based on the language independent specification.

Rationale: The STARS program is oriented to the provision of technology that supports the development of Ada software systems and is committed to the use of Ada in the development of software that constitutes that technology.

4.2.7 ALOAF Conformance Criteria

Aim: The ALOAF will define classes of conformance.

Rationale: The development of the ALOAF itself, of capabilities providing ALOAF services, and asset interchange support will progress over time. Classes of conformance will permit asset library systems flexibility and an evolutionary path in adherence to the ALOAF. Furthermore it may be the case that a non-STARS library would choose to participate only in asset interchange and/or in a limited form of asset interchange. Classes of conformance will provide that opportunity for non-STARS libraries.

Part II

The ALOAF Library Model

5 Data Modeling Concepts

The STARS ALOAF addresses three data modeling layers: the meta-model layer, the model layer and the data layer. Figure 2 depicts the relationships among these layers. In order to understand the STARS ALOAF, it is important to understand what these layers are, how they relate to asset libraries and how they relate to the STARS ALOAF.

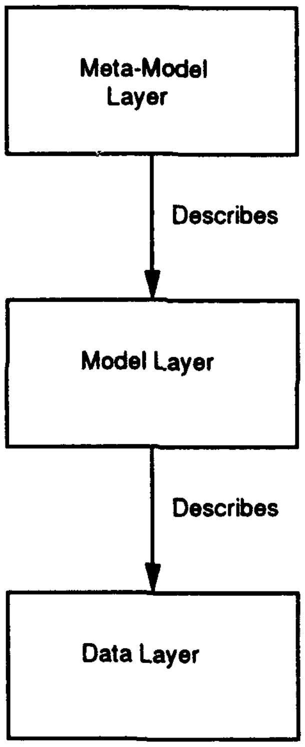


Figure 2: Data Modeling Layers

5.1 Data Modeling Layers

The three data modeling layers represent three levels of generality where the meta-model layer is the most general and the data layer is the least general. The meta-model layer consists of various data modeling techniques. ERA modeling, data flow modeling, relational modeling and object oriented modeling are all examples of data modeling techniques. An individual data modeling technique, referred to as a meta-model, permits the definition of a class of data models; such a class may be used to describe data for a wide range of applications or organizations.

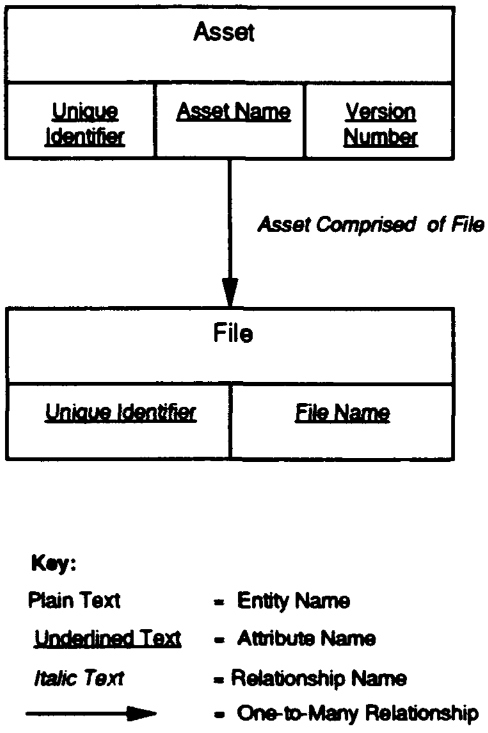


Figure 3: ERA Data Model

The model layer consists of data models. A data model conforms to a meta-model and describes data that is specific to an application or organization. The terms "schema" and "information model" are synonyms for data model. Figure 3 shows a simple example of a data model conforming to the ERA meta-model.[1]

The data layer consists of data such as the integer value 7, the character value 'A' or the string value "Hello World". The data in this layer are organized by and conform to data models in the model layer. Figure 4 shows a simple example of data conforming to the data model given in Figure 3.

5.2 Relationship to Asset Libraries

An asset library maintains a collection of assets and provides one or more classification schemes that a reuser may employ to locate an individual asset in that collection. An asset library may also provide a reuser with information about each asset, such as the creation date of or an abstract for each asset. All of the information that an asset library maintains about its assets, both information to support classification and information provided directly to a reuser, is described by the library's data model. In turn, an asset library's data model is described by its meta-model. Figure 1 illustrates the roles of the meta-model and the data model within an asset library. A library's meta-model and data model fit, respectively, into the meta-model and model layers described above; the information that a library maintains about its assets fits into the data layer.

[1]Note that the choice of the ERA meta-model for this example and, consequently, for the example in Figure 4 is for illustrative purposes only. The reader should not infer STARS ALOAF support for the ERA meta-model.

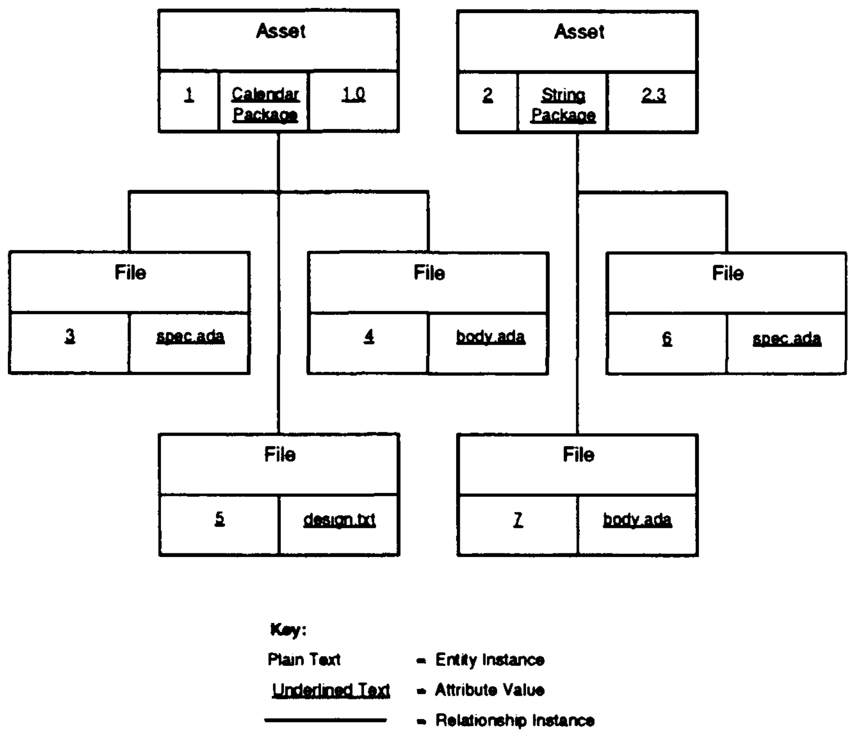


Figure 4: ERA Example Data

5.3 Relationship To STARS ALOAF

Asset library data models are ordinarily library specific and may differ greatly from one asset library to the next. Indeed, some asset library systems, such as Unisys's Reusability Library Framework (RLF)[SWT89] and SAIC's Asset Management System (AMS)[CD90], are generic and may be instantiated on a wide variety of data models. Also, the data models of domain-specific asset libraries are tailored to particular domains. The STARS ALOAF must define standards that accommodate these diverse data modeling needs. To do this, the STARS ALOAF includes a standard meta-model (the ALOAF Meta-Model, described in section 7.1) that permits the definition of a wide range of library data models. There is considerable precedence for the use of a standard meta-model in support of service definitions and data interchange mechanisms. The Portable Common Tool Environment (PCTE) [PCT90], A Tool Integration Standard (ATIS) [ATI90] and the ANSI Information Resource Dictionary System (IRDS) [IRD88] each define services which manipulate data conforming to a standard meta-model. Likewise, the CASE Data Interchange Format (CDIF) [CDI91a] and ANSI IRDS each support data formats based on standard meta-models.

The STARS ALOAF also includes a standard data model (the Common Data Model for Asset Interchange, described in section 7.2). The standard meta-model supports asset interchange between asset libraries with different data models; the standard data model provides a basis of communication for those libraries. That is, the standard data model describes basic information about assets that can be used to identify assets and to generate catalog entries for assets. The only constraint that the standard data model imposes on ALOAF-conformant asset libraries is that they may not export a data model that conflicts with the standard data model. This is a minor constraint, however, since the standard data model only describes a limited amount of information.

6 Service Model

The ALOAF service model defines the categories of services that individual asset libraries make available to reuse tools. The service model describes the relationships of individual services within a category as well as relationships across category boundaries. The service model provides the uniform interfaces (and associated terminologies, notations, and descriptions) that reuse tool developers and asset library framework suppliers use to communicate with each other.

The ALOAF service model is analogous in some respects to the draft ECMA Reference Model (RM), version 4.0 [Ear90]. The ECMA RM allows the interfaces among the major SEE services and the interfaces to individual tools within a SEE to be specified. A goal is to allow related services to be covered by one (or a small number of) standards instead of having each service defined by its own document. The ECMA RM also provides a means of locating an individual service among the complex jumble of services a complete SEE might offer; this serves as an aid to comparing and assessing different SEEs for how well they support portability and interoperability among any tools built on them.

Figure 5 corresponds to the well-known ECMA "toaster model" (see [Ear90], figure 1), providing a top level summary of the base service categories provided by the underlying SEE upon which ALOAF services will be implemented. Each of these SEE services is defined in section 5 of [Ear90].

The ECMA RM groups the SEE services it describes into the following categories:

- data repository services,
- data integration services,
- tools,
- task management services,
- message services, and
- user interface.

The ALOAF provides services within the framework of a host SEE, or more basically a host operating system, for a class of applications – specifically asset libraries and the tools that are designed to cooperate with the asset libraries. As such, a substrate of services is presumed to be available within the host environment to enable the implementation of the higher level ALOAF services. For more primitive environments that contain fewer services, the implementation of an ALOAF-conformant library system will require considerable effort. Within richer environments such as those based on an object management system, implementing such a library will likely be significantly easier.

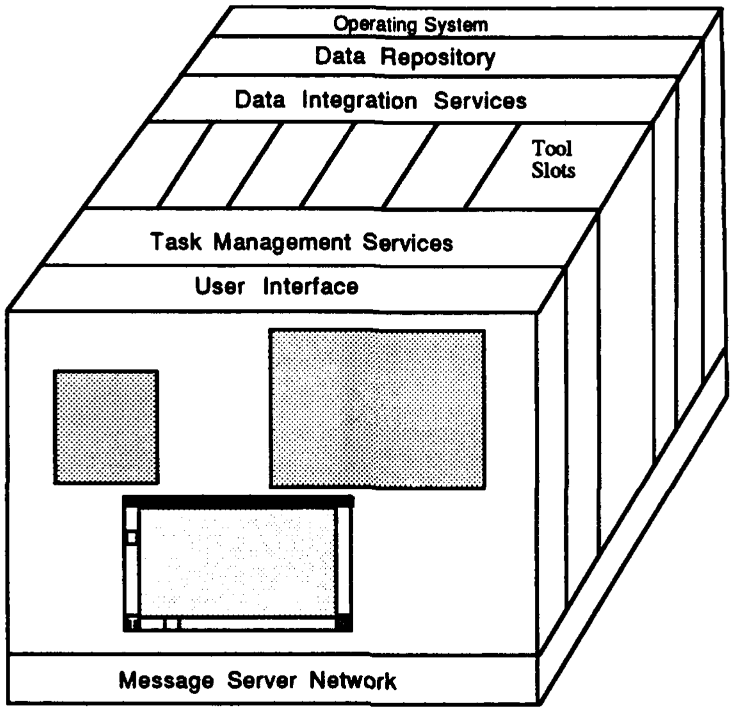


Figure 5: ECMA Toaster Model

Certain categories of services illustrated in figure 5 contain services that can be used directly without modification or tailoring. The user interface and task management service categories provide specific examples. The ALOAF does not require any reuse-specific versions of such services. In particular, the ALOAF service model does not define any user interface service "look and feel" because that is the responsibility of the developers of asset library tools that will make use of user interface capabilities provided by the host environment. The ECMA task management category contains the services more commonly identified as relating to process management. Other STARS tasks are addressing the creation and enactment of reuse-based processes which can be implemented using the process services provided within the host SEE. Thus, the ALOAF will not specify any reuse-specific process-related services.

Another area where ALOAF services can be expected to use the services provided by the host environment is the message service category. Seamless operations among asset libraries distributed over a network will require that communications protocols be established for how individual asset descriptions, asset library commands, and other interactive responses are transmitted over these networks. While it is likely that existing SEE message services may be able to provide these protocols, it may be necessary for higher level asset library protocols to be developed that directly address asset library semantic information. Such tailored specifications, if required, would be identified as extended services within the ALOAF.

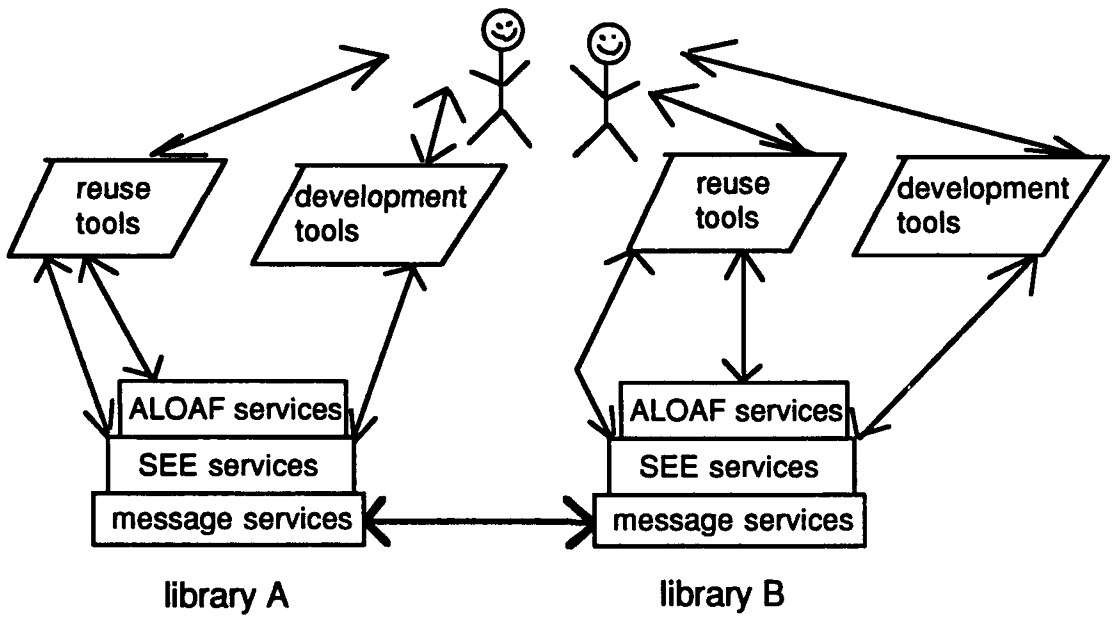


Figure 6: Relationships Among SEE and ALOAF Services

6.1 ALOAF Service Categories

The (assumed) SEE services described in the next section provide the foundation on which the ALOAF services described in this section are built. Figure 6 provides a simple diagram that shows this relationship among these services, the user tools that are built on them, and the (re)users that employ them. Note how the ALOAF is built on top of the SEE services, but not vice versa. Figure 6 can be seen as providing a two dimensional "slice" through the three dimensional figure 5.

The ALOAF services are organized into categories identified by the following list:

- library management,
- data model,
- asset description,
- session,
- query,
- asset location,
- metrics, and
- access control.

Each of the ALOAF service categories are briefly described below. Some of these ALOAF services may be provided entirely by functionality built-in to the underlying SEE's framework services, while others may be provided by SEE tools that communicate with the other services/tools through the message services and use SEE (and possibly other ALOAF) services. Note that "SEE tools" in this context refers to entities residing in ECMA tool slots (figure 5) and is not to be confused with the "library tools" shown in figure 1 in section 1, which are interpreted strictly as clients, and not as providers, of ALOAF services.

6.1.1 Library Management Services

This category includes services for managing and manipulating a library within its operating environment, carrying with it the structure of an implicit or explicit data model. This category provides the services to cause a new asset library to be created, or an existing library to be deleted as well as opening a library for modification and closing it after modifications are completed.

6.1.2 Data Model Services

This category includes services to manage and manipulate library data models in accordance with a particular meta-model, as discussed earlier in section 5. A library data model defines the structure of the information used to describe each library asset in order to organize, categorize, search, understand, evaluate, extract, adapt, and integrate the asset. The library's classification scheme is part of its data model. Data model services include general classes of operations such as read model element (e.g., class and attributes), write model element, update model element, delete model element, and import and export model.

6.1.3 Asset Description Services

The services in this category manage and manipulate individual asset descriptions. Assets are catalogued in the library in the form of asset descriptions represented as instances of classes in the library data model. The asset description services will allow descriptions to be read/written/updated/deleted, and will allow asset descriptions to be imported from and exported to other libraries.

6.1.4 Session Services

Session services manage connections to asset libraries. Such services support users as well as tools operating on behalf of users to, for example, transfer assets and asset descriptions between sites. This category includes any services associated with establishing user identity and access rights and with managing the resulting connection with the library. Session services support record keeping, user and system notifications, and similar functions associated with use of the library.

6.1.5 Query Services

The goal of query services is to provide effective means for a variety of users to identify the assets they want using flexible query mechanisms. The query services are the mechanisms used to support traversal of a library in the context of its data model to find the set of assets that best match the user's criteria. A query service provides an algebra on the possible asset description attributes. Queries may be iterative, with a user supplying successively stronger criteria to refine a search until a set of assets of interest is identified.

6.1.6 Asset Location Services

Once a user has identified assets of interest, the asset location services provide the information needed to obtain the assets. Underlying SEE services may be used to examine an asset and to retrieve it in a [re]usable form. Access to an asset may be more restrictive than query and identification and/or require additional steps.

6.1.7 Metrics Services

Libraries will generally be expected to collect library metrics information and make some portion of the information available to users and administrators. Reuse tools which provide reports on asset library usage and manage the change logs on the data models, asset descriptions, and assets will need corresponding services. The metrics services category includes services to provide metrics on the structure and population of the asset library. Services which control the collection and disposition of statistical information are also in this category.

6.1.8 Access Control Services

The access control services manage the information used to determine which ALOAF data and services a user is permitted to access. Access control services help maintain the wide variety of restrictions that may be placed on assets and on users either legally or by policy. The access controls may be different from traditional SEE Access Control Lists (ACLs). These ALOAF services may provide access controls at a very fine grain (such as controls on individual model attributes) or very large grain (such as reuse roles).

6.2 Assumed SEE Services

This section of the ALOAF service model identifies some SEE services that may be provided by an underlying SEE. Many ALOAF services will depend upon a SEE's services and so it is important that this ALOAF service model indicate the interrelationships among these services. The details of the functionality provided by these SEE services are outside the scope of this document.

Data Storage Service

Data storage services are associated with actual storage and retrieval of objects and information about objects. Object descriptions are the explicitly declared characteristics of entities ("attributes" or "properties"). "Information about objects" covers information needed for a tool to perform effectively, but not formally declared as an object attribute. A size in bytes of an object and its sub-objects (taken to the lowest level) might be one example.

Relationship Service

Relationship services are concerned with relating objects and other entities. ECMA defines a service dealing with defining and maintaining typed, attributed "relationships" in the formal E-R sense. ALOAF use of this Relationship Service is intended to be more general, including informal and temporary groupings of objects.

Name Service

Name services translate among system-generated unique identifiers ("surrogates" in the ECMA RM) and user-mnemonic names. Surrogates should be visible to tools, which may or may not make them visible to users; however, for most purposes, use of a name in association with environmental context should produce the same effect as use of the surrogate.

Location Service

Location services support distribution of the SEE and its objects.

Data Transaction Service

Data transaction services are associated with ensuring that work is performed in complete atomic units ("transactions").

Concurrency Service

Concurrency services are required to control simultaneous access to objects.

Process Support Service

Process support services are: "the basic support mechanisms for active entities. They provide mechanisms to support them while they are and are not executing and mechanisms to monitor them while they are." [Ear90].

Archive Service

Archive service: "carries out a mapping between the online storage and offline storage of entities. A placeholder may represent the entity in online storage, while the entity is archived offline." [Ear90].

Backup Service

Backup services provide the means to restore service following media failure. This could include reaccomplishing committed transactions since the last backup, using transaction logs or journals.

Data Transformation/Interchange Service

Data transformation/interchange services support two-way translation between objects in the repository and an agreed-on interchange format. This would include model description translation, as well as translations to and from various formats for internal storage.

History Service

History services are the accounting functions that make available information about changes of state of entities (broadly scoped to include states such as "popularity" (number of times queried/extracted) and "quality" (profile of problem reports). This service is probably closely related to transaction logging and journalling, but should provide the hooks for administrators to control amount and type of logging.

Version Service

Version services are the core functions of applying a version identification policy to entities in an object/file system.

Security Service

Security services are the subset of overall SEE security measures aimed at restricting access to objects and their services.

Tool Registration Service

Tool registration services are probably not separate services per se, but possibly some interaction in order to support security policies and tool usage policies (which tools can access which object types).

Part III

The ALOAF Specifications

7 Data Modeling and Asset Interchange

The concepts of data modeling and asset interchange are inextricably linked. Asset interchange is the act of transferring one or more assets from one asset library to another. However, as discussed in section 5.2, asset libraries, in addition to managing collections of assets, maintain information about their assets. The information that a library maintains for each asset represents a significant amount of cataloging labor. In short, this information is valuable. The STARS ALOAF supports the view that an asset interchange mechanism must be capable of transferring both assets and information about assets. To reflect this view, the Asset Interchange Specification must include a library independent representation for information about assets. In turn, this representation requires a library independent technique for describing information about assets. The ALOAF Meta-Model, described in section 7.1, satisfies this requirement. The Asset Interchange Specification's dependence on the ALOAF Meta-Model forms the relationship between data modeling and asset interchange.

The Asset Interchange Specification embodies both a short term approach and a long term approach to asset interchange. The goal of the short term approach is to provide a rudimentary asset interchange capability as quickly as possible. The goal of the long term approach is to develop a powerful and flexible asset interchange capability that will effectively serve the needs of distributed heterogeneous asset libraries. The short term approach is based upon a standard data model, termed the Common Data Model for Asset Interchange. The fundamental drawback to this approach is that no standard data model can accommodate the diverse data modeling needs of asset libraries. In the short term, however, constructing an asset interchange approach around a standard data model has one significant advantage: it can be implemented quickly.

The long term approach to asset interchange is based upon a standard meta-model, the ALOAF Meta-Model, and an asset library independent representation of library data models and data, termed the Asset Interchange Data Format. The drawback of the short term approach is the strength of the long term approach. The ALOAF Meta-Model and the Asset Interchange Data Format allow asset libraries to exchange information conforming to a wide range of library data models. The Common Data Model, as a standard data model, also plays a role in the long term approach, as described in section 5.3. Ideally, the Asset Interchange Data Format will be derived from existing or evolving data interchange standards. This will ensure that the format will remain suitable for asset interchange in the future as well as in the present. Because of this, the evaluation of data interchange standards plays a significant role in the long term approach.

This version of the STARS ALOAF document includes major contributions to both asset interchange approaches. The Common Data Model for Asset Interchange, described in section 7.2, and the Asset Interchange Language, described in appendix D[2], are an implementation of the short

[2]The Asset Interchange Language provides a simple technique for representing data that conforms to the Common Data Model. This language was not derived from any existing or emerging standards and was developed solely as a stop-gap measure to support a rudimentary asset interchange capability.

term approach. The Common Data Model is also a first attempt at satisfying the standard data model needs of the long term approach. The ALOAF Meta-Model, described in section 7.1, is the cornerstone of the long term approach. This meta-model provides a standard technique for constructing library data models and supports the development of a library independent representation for those models. The ALOAF Meta-Model and the standards evaluation activity, described in section 7.3, are the first steps towards the specification of the Asset Interchange Data Format.

7.1 The Meta-Model

An asset description is not simply an arbitrary collection of data about an asset. Each asset library includes a data model which defines the structure and content of that library's asset descriptions. Because of this, asset libraries need certain fundamental data modeling capabilities. The ALOAF Meta-Model defines these capabilities. That is, this meta-model provides the basic constructs and rules that are used in the creation and modification of library data models. The ALOAF Meta-Model does not define the structure or content of the information maintained by asset libraries; rather, it provides a means for defining that structure and content.

A standard library meta-model offers three benefits. First, it can be used as the foundation for a library-independent representation of library data models. In turn, this representation can be used to support a sophisticated asset interchange capability. Second, a meta-model supports the development of data model independent service specifications. Such specifications can be used as an open architecture standard for asset libraries. Finally, a meta-model provides some fundamental concepts and terminology for individuals involved in the development of library data models.

The ALOAF Meta-Model is divided into two sections: the Core Meta-Model and Meta-Model Extensions. The Core Meta-Model includes the basic modeling techniques that must be supported by all ALOAF implementations. Meta-Model Extensions are important data modeling techniques that, due to considerations of technical difficulty and present immaturity of evolving standards, need not be supported by all ALOAF implementations. The rationale for a particular extension is provided in the section which describes that extension.

7.1.1 The Core Meta-Model

The ALOAF Core Meta-Model defines two constructs, classes and relationships, and two concepts, specialization and instantiation. The two constructs are the basic components or "building blocks" of library data models. Specialization is a relationship that is fundamental to the structure of a library data model. Instantiation is the technique by which the constructs of asset descriptions, objects and links, are created from the constructs of library data models, classes and relationships. Objects and links will be described in more detail in section 7.1.1.4.

Note that the Core Meta-Model's treatment of relationships is tentative and is under consideration by the participants of the ALOAF working group.

7.1.1.1 Classes A class models a set of objects that share a common structure.[3] The attributes of a class define that common structure; each attribute defines some individually accessible part of that structure. A class may have an unlimited number of attributes. Each attribute shall belong to exactly one class.

The information needed to define a class is:

Class Name A mnemonic, human-readable identifier for the class. This name must be unique across the names of all the classes in the library data model.

Parent Class The class identifier of the parent class. For a description of parent classes see section 7.1.1.3.

The information needed to define each attribute of a class is:

Attribute Name A mnemonic, human-readable identifier for the attribute. This name must be unique across the names of all the attributes of the class.

Attribute Value Type The type of data that will be supplied for the attribute when the class is instantiated. The valid attribute value types and the possible values for those types are:

Value Type	Possible Values
Boolean	TRUE or FALSE
Integer	An integer number
Float	A floating point number
Time	A date and time of day
String	A sequence of ASCII characters

7.1.1.2 Relationships A relationship between classes models the associations, or links, that may be created between instances of those classes. A class may participate in an unlimited number of relationships. A relationship shall emanate from exactly one class (termed the source class) and shall target exactly one class (termed the destination class). The source and destination classes for a relationship may be the same class.

The information needed to define a relationship is:

Relationship Name A mnemonic, human-readable identifier for the relationship. This name must be unique amongst the names of all relationships that have the same source and destination classes.

Source Class The class identifier for the class that is the source of the relationship.

[3]The typical object-oriented view of a class, as stated in [Boo91], is "a set of objects that share a common structure and behavior." The ALOAF Meta-Model, however, is a meta-data model; the class construct defined in this meta-model does not model object behavior.

Target Class The class identifier for the class that is the destination of the relationship.

Lower Bound The minimum number of links (instances of the relationship) that may emanate from an object of source class. The lower bound may never be greater than the upper bound.

Upper Bound The maximum number of links (instances of the relationship) that may emanate from an object of source class. The upper bound may never be less than the lower bound.

7.1.1.3 Specialization The classes of a library data model are formed into a specialization hierarchy. That is, each class in a library data model participates in the specialization, or parent/child, relationship. In this relationship, a class shall have exactly one parent class (except for the root class, see below) and may have an unlimited number of child classes. The hierarchy formed by the specialization relationship is acyclic; a class may never be the parent of any of its ancestors (e.g. its parent, its parent's parent, etc.).

The specialization relationship has two semantic implications. First, a class inherits the attributes of all its ancestor classes. Inherited attributes, other than the fact that they are inherited, are indistinguishable from attributes specifically defined for a particular class. Second, a class may participate in any relationship in which its ancestors may participate. The rules regarding the names of attributes and relationships also apply to inherited attributes and relationships.

Every library data model includes a special class known as the root class. This class has no parents and is an ancestor of every class in the specialization hierarchy. The class name, class identifier and attributes of the root class are not defined by this meta-model.

Note that the specialization concept defined in the ALOAF Core Meta-Model implies single inheritance. Multiple inheritance is addressed as a Meta-Model Extension.

7.1.1.4 Instantiation An asset description is created by instantiating classes and relationships An instance of a class is referred to as an object. A class may be instantiated by an unlimited number of objects. Each object shall be an instance of exactly one class. The information needed to instantiate a class is:

Object Identifier A unique identifier for the object amongst all the objects maintained by an asset library.

Attribute Values A default initial value for each attribute associated with the class. The default initial value for a particular attribute depends upon its value type, as follows:

Value Type	Default Initial Value
Boolean	FALSE
Integer	0
Float	0.0
Time	00:00:00, January 1, 1970
String	"" (empty string)

An instance of a relationship is referred to as a link. Barring violation of upper bound constraints, a relationship may be instantiated by an unlimited number of links. Each link shall be an instance of exactly one relationship. The information needed to instantiate a relationship is:

Link Identifier A unique identifier for the link amongst all the links maintained by an asset library.

Source Object The unique identifier for the source object of the link. The source object must be an instance of the source class defined by the relationship.

Destination Object The unique identifier for the destination object of the link. The destination object must be an instance of the destination class defined by the relationship.

Also, a relationship instantiation shall not violate the relationship's upper bound *constraint on the* source object. Symmetrically, when a link is deleted, the deletion shall not violate the relationship's lower bound constraint on the source object.

Note that the ALOAF Core Meta-Model specifies single instantiation. Each object shall be an instance of exactly one class. Multiple instantiation, where each object may be an instance of an unlimited number of classes, is addressed as a Meta-Model Extension.

7.1.2 Meta-Model Extensions

Multiple inheritance and multiple instantiation have been identified as useful meta-model extensions. Multiple inheritance extends the Core Meta-Model's inheritance capability to allow each class to have an unlimited number of parent classes. Multiple instantiation extends the Core Meta-Model's instantiation capability to allow each object to instantiate an unlimited number of classes. These extensions will be described in detail in Version 1.0 of the ALOAF document.

7.2 The Common Data Model for Asset Interchange

This section supports the short term approach to asset interchange by defining the Common Data Model for Asset Interchange. This data model describes information that is commonly maintained by a majority of STARS asset libraries.

This section also supports the long term approach to asset interchange by defining the basic information needed to identify and generate catalog entries for assets. In the long term, an asset library that conforms to the ALOAF must not export a data model that conflicts with the Common Data Model. Otherwise, an ALOAF-conformant importing library will misinterpret the semantics of data conforming to that conflicting model. It is important to understand, however, that the Common Data Model does not place any requirements on the internal data model of a conforming asset library. There is no requirement that a conforming asset library export its internal data model; it merely may not export a data model that conflicts with the Common Data Model.

Note that this version of the Common Data Model preceded the ALOAF Meta-Model and includes an implicit meta-model that is similar to but independent of the ALOAF Meta-Model. Version

1.0 of the ALOAF document will include a Common Data Model that conforms to the ALOAF Meta-Model.

7.2.1 The Representation of the Common Data Model

The Common Data Model will be represented as a class hierarchy. There are two reasons for this representation. First, the information maintained about an asset by each STARS asset library can be represented as a class hierarchy. In other words, this representation is sufficiently general. Second, due to the popularity of object oriented development, class hierarchies are widely understood and can be readily communicated to the software engineering community.

7.2.1.1 The Fundamentals A class defines a set of objects. The objects included in a class have similar behavioral characteristics and similar attributes. The Common Data Model uses a degenerate form of this definition of a class. Since the data model is a description of a data structure, only attributes are included in the data model's class definitions. The behavioral characteristics of these classes are irrelevant.

Classes can be organized into a hierarchy based upon a relationship among the classes. The Common Data Model's class hierarchy is based upon the inheritance (or "is a kind of") relationship. Every class in this class hierarchy inherits the attributes of its superclass. The Common Data Model does not require multiple class hierarchies based upon the inheritance relationship (i.e., multiple inheritance).

Attributes maintain the state of an object. The Common Data Model requires four types of attributes: string, text, date and link. String attributes simply maintain strings of ASCII characters. Text attributes maintain "2-dimensional" strings of ASCII characters (large blocks of text). Date attributes maintain calendar dates. Link attributes maintain relationships (other than the inheritance relationship) among classes. The relationships defined by link attributes are bi-directional and have specified cardinalities in each direction.

7.2.1.2 The Representation of a Class Hierarchy The Common Data Model's class hierarchy is represented using indented class names as follows:

```
Root Class Name
  Class Name 1
    Subclass Name 1
    Subclass Name 2
        .
        .
        .
    Subclass Name N
  Class Name 2
      .
      .
      .
```

```
Class Name N
```

This representation of a class hierarchy is easy to maintain in a text file and, for shallow hierarchies, is easy to read.

7.2.1.3 The Representation of a Class As stated in section 7.2.1.1, attributes are the main components of the Common Data Model's class definitions. As such, a class definition includes the location of the class in the hierarchy (i.e., its superclass and its subclass(es)) and a collection of attributes. The following is a template for the representation of a class (items enclosed in brackets are optional):

```
 CLASS NAME         :
[SUPERCLASS NAME    :]
[SUBCLASS NAME(S)   :]
 ATTRIBUTE NAME(S)  :
```

Note that the SUPERCLASS NAME information is optional because the root node in the hierarchy will not have a parent class. In similar fashion, the classes at the bottom of the class hierarchy will not need the SUBCLASS NAME(S) information.

7.2.1.4 The Representation of an Attribute To complete the definition of a class, the attributes of that class must be defined. A general attribute definition includes the following information:

Name The attribute's name

Type The attribute's type (string, text, date or link)

Mandatory Whether or not a value must be supplied for the attribute

Unique Whether or not the value for that attribute must be unique across an independent data set conforming to a particular data model (this information is replaced by cardinality information in link attributes)

A link attribute definition additionally includes:

Range Class The target class for the relationship specified by the link attribute

Inverse The name of the link attribute that specifies the inverse relationship

Cardinality The number of objects that the specified relationship can target (this information replaces uniqueness information)

¿From this information a template representation of an attribute definition can be created. This template is as follows (items enclosed in brackets are optional):

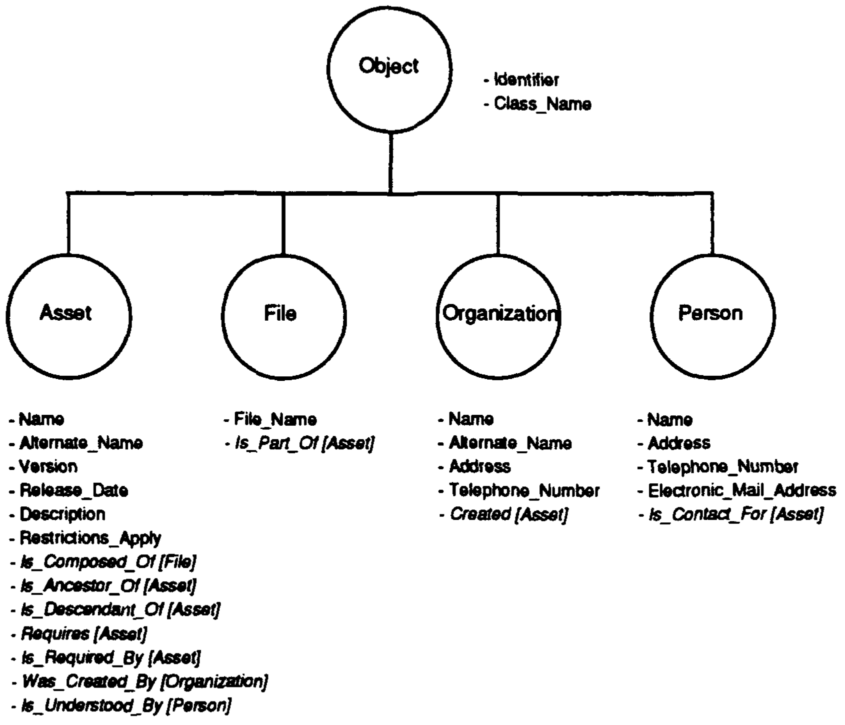


Figure 7: The Common Data Model

```
 NAME          :
 TYPE          :
 MANDATORY     :
[UNIQUE        :]
[RANGE CLASS   :]
[INVERSE       :]
[CARDINALITY   :]
```

7.2.1.5 Constraints Imposed by this Representation Any organization that attempts to create a data model using this representation will have three constraints imposed upon it by this representation. Each of these constraints relates to the integrity of the data model. First, attribute names must be unique. A class identifies each of its attributes using the attribute name. Second, class names must be unique. A link attribute identifies the range class for its relationship using the class name. Finally, the creating organization must be careful to specify link attributes in both directions.

7.2.2 The Model

The class hierarchy, classes and attributes of the Common Data Model are defined using the representations provided in sections 7.2.1.2, 7.2.1.3 and 7.2.1.4, respectively. A graphic representation of the Common Data Model is given in Figure 7

7.2.2.1 The Class Hierarchy

```
Object
  Asset
  File
  Person
  Organization
```

Although the root class of this hierarchy is named Object, this hierarchy is definitely oriented towards supporting the Asset class. The Object class was created in recognition of the fact that some of the information needed to support an asset is external to the Asset class. Objects from the remaining classes, File, Person and Organization, provide supporting information for assets.

7.2.2.2 The Classes

7.2.2.2.1 The Object Class

```
CLASS NAME        : Object
SUBCLASS NAME(S)  : Asset
                    File
                    Organization
                    Person
ATTRIBUTE NAME(S) : Identifier
                    Class_Name
```

This class represents the class of all objects that can be described by the Common Data Model. This class serves as the root class for the Common Data Model and, as such, all the other classes in the model inherit the attributes of this class.

7.2.2.2.2 The Asset Class

```
CLASS NAME        : Asset
SUPERCLASS NAME   : Object
ATTRIBUTE NAME(S) : Name
                    Alternate_Name
                    Version
                    Release_Date
                    Description
                    Restrictions_Apply
                    Is_Composed_Of
                    Is_Ancestor_Of
                    Is_Descendant_Of
                    Requires
```

```
                                  Is_Required_By
                                  Was_Created_By
                                  Is_Understood_By
```

This class represents the class of all assets that can be described by the Common Data Model. The physical representation of an asset is a collection of zero or more files. Additionally, an asset may reference people, organizations or other assets for descriptive context.

7.2.2.2.3 The File Class

```
CLASS NAME        : File
SUPERCLASS NAME   : Object
ATTRIBUTE NAME(S) : File_Name
                    Is_Part_Of
```

This class represents the class of all files that can be described by the Common Data Model. An asset is composed of zero or more files. The Common Data Model does not attempt to specify the format, content or access mechanisms for these files. It is expected, however, that these files will conform to the POSIX standard.

7.2.2.2.4 The Organization Class

```
CLASS NAME        : Organization
SUPERCLASS NAME   : Object
ATTRIBUTE NAME(S) : Name
                    Alternate_Name
                    Address
                    Telephone_Number
                    Created
```

This class represents the class of all organizations that can be described by the Common Data Model. Assets are created by organizations and this class allows us to capture information about those organizations and to link assets with that information.

7.2.2.2.5 The Person Class

```
CLASS NAME        : Person
SUPERCLASS NAME   : Object
ATTRIBUTE NAME(S) : Name
                    Address
                    Telephone_Number
                    Electronic_Mail_Address
                    Is_Contact_For
```

This class represents the class of all people that can be described by the Common Data Model. Assets are understood by people and this class allows us to capture information about those people and to link assets with that information.

7.2.2.3 The Attributes

7.2.2.3.1 The Address Attribute

```
NAME        : Address
TYPE        : String
MANDATORY   : False
UNIQUE      : False
```

This attribute allows each organization and person described by the Common Data Model to have an address.

7.2.2.3.2 The Alternate_Name Attribute

```
NAME        : Alternate_Name
TYPE        : String
MANDATORY   : False
UNIQUE      : False
```

This attribute allows each asset and organization described by the Common Data Model to have an alternate (possibly abbreviated) name. This attribute is intended as a human-readable identifier for an asset or organization.

7.2.2.3.3 The Class_Name Attribute

```
NAME        : Class_Name
TYPE        : String
MANDATORY   : True
UNIQUE      : False
```

This attribute specifies that each object described by the Common Data Model will identify the class under which it was instantiated.

7.2.2.3.4 The Created Attribute

```
NAME        : Created
```

```
TYPE        : Link
MANDATORY   : False
RANGE CLASS : Asset
INVERSE     : Was_Created_By
CARDINALITY : N
```

This attribute allows each organization described by the Common Data Model to serve as the creating organization for one or more assets.

7.2.2.3.5 The Description Attribute

```
NAME        : Description
TYPE        : Text
MANDATORY   : False
UNIQUE      : False
```

This attribute allows each asset described by the Common Data Model to have an abstract.

7.2.2.3.6 The Electronic_Mail_Address Attribute

```
NAME        : Electronic_Mail_Address
TYPE        : String
MANDATORY   : False
UNIQUE      : False
```

This attribute allows each person described by the Common Data Model to have an electronic mail address.

7.2.2.3.7 The File_Name Attribute

```
NAME        : File_Name
TYPE        : String
MANDATORY   : True
UNIQUE      : True
```

This attribute specifies that each file described by the Common Data Model will maintain a unique POSIX file name.

7.2.2.3.8 The Identifier Attribute

NAME	:	Identifier
TYPE	:	String
MANDATORY	:	True
UNIQUE	:	True

This attribute specifies that each object described by the Common Data Model will have a unique identifier.

7.2.2.3.9 The Is_Ancestor_Of Attribute

NAME	:	Is_Ancestor_Of
TYPE	:	Link
MANDATORY	:	False
RANGE CLASS	:	Asset
INVERSE	:	Is_Descendant_Of
CARDINALITY	:	N

This attribute allows each asset described by the Common Data Model to identify one or more descendant assets.

7.2.2.3.10 The Is_Composed_Of Attribute

NAME	:	Is_Composed_Of
TYPE	:	Link
MANDATORY	:	False
RANGE CLASS	:	File
INVERSE	:	Is_Part_Of
CARDINALITY	:	N

This attribute allows each asset described by the Common Data Model to be composed of one or more files.

7.2.2.3.11 The Is_Contact_For Attribute

NAME	:	Is_Contact_For
TYPE	:	Link
MANDATORY	:	False
RANGE CLASS	:	Asset
INVERSE	:	Is_Understood_By
CARDINALITY	:	N

This attribute allows each person described by the Common Data Model to serve as the contact for one or more assets.

7.2.2.3.12 The Is_Descendant_Of Attribute

```
NAME        : Is_Descendant_Of
TYPE        : Link
MANDATORY   : False
RANGE CLASS : Asset
INVERSE     : Is_Ancestor_Of
CARDINALITY : N
```

This attribute allows each asset described by the Common Data Model to identify one or more ancestor assets.

7.2.2.3.13 The Is_Part_Of Attribute

```
NAME        : Is_Part_Of
TYPE        : Link
MANDATORY   : True
RANGE CLASS : Asset
INVERSE     : Is_Composed_Of
CARDINALITY : N
```

This attribute specifies that each file described by the Common Data Model must be a component of one or more assets.

7.2.2.3.14 The Is_Required_By Attribute

```
NAME        : Is_Required_By
TYPE        : Link
MANDATORY   : False
RANGE CLASS : Asset
INVERSE     : Requires
CARDINALITY : N
```

This attribute allows each asset described by the Common Data Model to be relied upon by one or more other assets.

7.2.2.3.15 The Is_Understood_By Attribute

```
NAME        : Is_Understood_By
TYPE        : Link
MANDATORY   : False
RANGE CLASS : Person
INVERSE     : Is_Contact_For
CARDINALITY : N
```

This attribute allows each asset described by the Common Data Model to identify one or more points of contact.

7.2.2.3.16 The Name Attribute

```
NAME        : Name
TYPE        : String
MANDATORY   : True
UNIQUE      : False
```

This attribute specifies that each asset, organization and person described by the Common Data Model will have a name. The name attribute is intended as a human-readable identifier for an asset, organization and person.

7.2.2.3.17 The Release_Date Attribute

```
NAME        : Release_Date
TYPE        : Date
MANDATORY   : False
UNIQUE      : False
```

This attribute allows each asset described by the Common Data Model to have a release date. The release date is intended to identify the date the asset was released by the creating organization.

7.2.2.3.18 The Requires Attribute

```
NAME        : Requires
TYPE        : Link
MANDATORY   : False
RANGE CLASS : Asset
INVERSE     : Is_Required_By
CARDINALITY : N
```

This attribute allows each asset described by the Common Data Model to depend upon one or more other assets.

7.2.2.3.19 The Restrictions_Apply Attribute

```
NAME        : Restrictions_Apply
TYPE        : Text
MANDATORY   : False
UNIQUE      : False
```

This attribute allows each asset described by the Common Data Model to have a statement of the legal restrictions imposed upon any part of that asset.

7.2.2.3.20 The Telephone_Number Attribute

```
NAME        : Telephone_Number
TYPE        : String
MANDATORY   : False
UNIQUE      : False
```

This attribute allows each organization and person described by the Common Data Model to have a telephone number.

7.2.2.3.21 The Version Attribute

```
NAME        : Version
TYPE        : String
MANDATORY   : False
UNIQUE      : False
```

This attribute allows each asset described by the Common Data Model to have a version identifier. The version identifier is intended as a human-readable identifier. This identifier would allow individuals to distinguish among variations and revisions of the same asset.

7.2.2.3.22 The Was_Created_By Attribute

```
NAME        : Was_Created_By
TYPE        : Link
MANDATORY   : False
RANGE CLASS : Organization
INVERSE     : Created
CARDINALITY : N
```

This attribute allows each asset described by the Common Data Model to identify one or more creating organizations.

7.2.3 Rationale for the Common Data Model

The definition of the Common Data Model was driven by three basic requirements. First, the model should not include any library-specific asset classification information. Second, the model should use as few classes and attributes as possible. Third, the model should be extensible. The basis for each of these requirements is discussed in the following paragraphs.

Asset classification information is always dependent upon a particular classification scheme. It is expected that the asset libraries conforming to the ALOAF will employ a wide variety of classification schemes for their assets. As such, asset classification information will vary (possibly wildly) from one asset library to another. It seems unlikely that the ALOAF will be able to require asset libraries to export a common classification scheme. Therefore, since the Common Data Model includes only information that is common across a variety of asset libraries, it should not include library-specific classification information.

In the long term, one requirement placed upon asset libraries conforming to the ALOAF is that they should not export a data model that conflicts with the Common Data Model. The ALOAF should not make it difficult for asset libraries to satisfy this requirement. One way to achieve this goal is to specify a data model that uses as few classes and attributes as possible. By specifying such a "low profile" data model, the probability of interfering with other data models is reduced.

The information included in the Common Data Model is currently maintained by a majority of asset libraries that have been built using STARS asset library mechanisms. However, it is expected that the ALOAF will be addressing the needs of a larger collection of asset libraries. At some point the common information needs of those libraries may exceed the capabilities of the Common Data Model. In order to satisfy any future common information needs, the data model should be extensible.

7.2.3.1 The Rationale for the Class Hierarchy In order to understand the Common Data Model's class hierarchy we need to begin with its purpose. The Common Data Model is meant to facilitate the interchange of assets and descriptive information about assets among asset libraries. In turn, the purpose of an asset library is to maintain an electronic catalog of reusable software assets and to provide a mechanism for rapidly accessing any of the cataloged assets. From this we can begin to understand the type of information that is going to be interchanged. Ultimately, reusable software assets are collections of files.[4] The collection of files comprising any given asset is generated by software engineers using software development tools. At some point this collection of files is deemed reusable and is cataloged in an asset library. Descriptive information about an asset may be included in an asset library's catalog. This information typically includes some description of the organization(s) that created the asset and some description of the person (or people) that understand the asset. In accordance with this view of reusable software assets and asset libraries, the classes `Asset`, `File`, `Organization` and `Person` have been included in the Common Data Model. Each of these classes is discussed in the next section.

7.2.3.2 The Rationale for the Classes and the Attributes The `Object` class was created to support classes external to the `Asset` class. A beneficial by-product of this class is that it also supports extensibility. Additional subclasses of the `Object` class, modeling objects that are external to any of the existing subclasses, can easily be added. The attributes of the `Object` class provide a unique identifier and a class name for all objects in the data model. The `Identifier` attribute supports link attributes. That is, the attribute value of a link attribute is the unique identifier of the target object. The `Class_Name` attribute supports importing asset libraries. A library that imports data conforming to this model will be able to determine an object's class based on its

[4]In more advanced software development environments, assets may be composed of finer grained components. The principles that apply to files apply to those finer grained components as well.

Class_Name attribute value.

The physical representation of a reusable software asset is a collection of files. The abstract representation of an asset is defined by the **Asset** class. Each of the following attributes describe an asset:

- The **Name**, **Alternate_Name** and **Version** attributes provide reusers with a convenient mechanism for referring to an asset.
- The **Release_Date** attribute provides reusers with the age of an asset.
- The **Description** attribute provides reusers with a broad overview of an asset that cannot be obtained from the classification information.

Each of the following attributes describe references to assets:

- The **Is_Ancestor_Of** and **Is_Descendant_Of** attributes identify the ancestors and descendants, respectively, of an asset.
- The **Requires** and **Is_Required_By** attributes identify an asset's location in a dependency hierarchy.

Each of the following attributes describe references to external objects that support an asset:

- The **Is_Composed_Of** attribute identifies the physical representation of an asset.
- The **Was_Created_By** attribute identifies the organization(s) that created an asset.
- The **Is_Understood_By** attribute identifies the person (or people) that understand an asset.

Each of these attributes describes information about an asset that is commonly available from a majority of the STARS asset libraries. Additionally, the **Is_Composed_Of** attribute supports importing asset libraries by identifying the physical representation of an asset.

An asset is composed of files. The **File** class represents external files stored in the POSIX file system. The **Is_Part_Of** attribute and its inverse, the **Is_Composed_Of** attribute of the **Asset** class, describe a relationship between files and assets. The **File_Name** attribute of this class describes a unique identifier for external files (i.e. a file name in the POSIX file system). Both of these attributes support importing asset libraries by linking an asset with the external files that are components of that asset.

Assets are created by organizations. The **Organization** class represents external organizations such as corporations or government agencies. The **Name**, **Alternate_Name**, **Address** and **Telephone_Number** attributes describe information that refers to an external organization. This information is commonly available from a majority of the STARS asset libraries. The **Created** attribute and its inverse, the **Was_Created_By** attribute of the **Asset** class, support importing asset libraries by describing the relationship between assets and organizations.

Assets are understood by people. The **Person** class represents actual people such as software engineers or program managers. The **Name**, **Address**, **Telephone_Number** and **Electronic_Mail_Address** attributes describe information that refers to an actual person. This information is commonly available from a majority of the STARS asset libraries. The **Is_Contact_For** attribute and its inverse, the **Is_Understood_By** attribute of the **Asset** class, support importing asset libraries by describing a relationship between assets and people.

7.3 Evaluation of Standards

This section supports the long term approach to asset interchange by describing the status of the standards evaluation activity. This activity is primarily oriented towards asset interchange. The goal of this activity is to determine if there are any existing or maturing standards that include data interchange capabilities equivalent to those capabilities required by the asset interchange long term approach. Currently, this activity has determined that three standards, summarized in the following three subsections, include capabilities similar to those required by the long term approach. Other standards have also been evaluated and the results of those evaluations are described in section 7.3.4. The initial conclusions of this activity are presented in section 7.3.5.

7.3.1 Summary of the Data Interchange Capabilities of ANSI IRDS

The ANSI Information Resource Dictionary System (IRDS) is American National Standard X3.138-1988 [IRD88]. This standard defines the requirements for a software tool that can be used to manage an Information Resource Dictionary (IRD). An IRD is a model of an organization's information resources. ANSI IRDS specifies a set of commands, the IRD-IRD Interface commands, that are used to transfer an IRD among conformant implementations of the standard. Note that when an IRD is transferred, the technique used to describe that IRD, the IRD Schema, must be transferred with that IRD. The ANSI IRDS Export-Import File Format, a draft proposed American National Standard, provides a representation for IRD Schemas and IRD's. The ANSI IRDS standard does not yet address the management of information resources, it only addresses the management of IRD's.

Approach An IRD is transferred when the exporting IRDS generates an export/import file and an importing IRDS reads and processes that file. The ANSI IRDS standard specifies IRD export and import commands that must be supported by a conformant IRDS. The rules governing IRD export and import are defined by these commands. The standard also specifies the content of the export/import file by defining the IRD Schema which is used to describe IRD's. The format of the export/import file is specified by the ANSI IRDS Export-Import File Format.

The ANSI IRDS standard defines the IRD Schema as an entity-relationship model consisting of the Minimal IRD Schema and, optionally, the Basic Functional IRD Schema. The Minimal IRD Schema supports administrative control of the IRD Schema and the IRD. All implementations of the standard must include the Minimal IRD Schema. The Basic Functional IRD Schema extends the Minimal IRD Schema and supports IRDS implementations by providing a "starter" technique for describing an IRD. The standard allows extensions of the Minimal IRD Schema and specifies commands that may be used to modify the IRD Schema. In general, an implementation is allowed to modify the IRD Schema, even the Basic Functional IRD Schema, as long as those modifications do not conflict with the Minimal IRD Schema. Currently, ANSI Technical Committee X3H4, the organization responsible for maintaining the ANSI IRDS standard, is working on the development of a standard IRD Schema that could be used to describe models from a wide range of subject areas. Subject areas under consideration include Data Flow modeling and Entity-Relationship modeling.

An export/import file consists of a header component, an IRD Schema component and an IRD component. The header component identifies the exporting IRDS and the specific IRD within that IRDS that was used to generate the export/import file. Additionally, the header component includes the IRD export command parameters and commentary from the file's creator. The IRD Schema component contains a partial description of the exporting IRDS' IRD Schema. This partial description is sufficient for an importing IRDS to determine IRD Schema compatibility. The IRD section contains the IRD being transferred. It may contain either the entire IRD conforming to the given IRD Schema or it may contain a, possibly null, subset of that IRD.

The format of an export/import file is based on ISO 8824, Information Processing Systems - Open System Interconnection - Specification of Abstract Syntax Notation One (ASN.1) and ISO 8825, Information Processing Systems - Open Systems Interconnection - Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1). The ANSI IRDS Export-Import Format provides an ASN.1 syntax used to describe the contents of an export/import file. The draft proposed standard also provides rules, beyond those provided by ISO 8825, for encoding that particular syntax.

Relevance to Asset Interchange ANSI IRDS and the ANSI IRDS Export-Import Format provide a reasonable mechanism for interchanging data models among asset libraries. The advantages to this mechanism are conformance to ANSI and ISO standards, support for multiple IRD Schemas, and the ability to represent any IRD Schema in the export-import file format. One disadvantage to this mechanism is that the export-import file format assumes interchange between conforming implementations of ANSI IRDS. Also, X3H4 is only just beginning work on a standard IRD Schema that supports multiple subject areas.

ANSI IRDS and the ANSI IRDS Export-Import File Format do not provide any mechanisms for interchanging data among asset libraries.

7.3.2 Summary of the CASE Data Interchange Format

The CASE Data Interchange Format (CDIF) standards family is a product of the Electronic Industries Association (EIA). This standards family is composed of three draft standards: EIA-PN2387 CDIF - Framework for Modeling and Extensibility [CDI91a], EIA-PN2329 CDIF - Standardized CASE Interchange Meta-Model [CDI91b] and EIA-PN2389 CDIF - Transfer Format Definition [CDI91c]. This standards family supports the interchange of models among CASE tools by providing a standard technique for describing models and a standard representation for models. This standards family does not address the interchange of data, it only addresses the interchange of models.

Approach A CDIF transfer occurs when an exporting tool generates a CDIF transfer file and an importing tool reads and processes that transfer file. The CDIF standards family does not specify interfaces for the importing or exporting tools. The standards family does specify the content and format of CDIF transfer files.

The content of a CDIF transfer file is specified by the Standardized CASE Interchange Meta-Model (hereafter referred to as the CDIF meta-model). The CDIF meta-model consists of a semantic meta-model and a presentation meta-model. The semantic meta-model can currently be used to describe the semantics of Entity-Relationship models, Data Flow models, and data typing schemes. The semantic meta-model is not limited to these subject areas, however. The CDIF Technical Committee, the component of the EIA that is responsible for the CDIF standards family, plans to extend the semantic meta-model to include additional subject areas such as State Transition Diagrams, Project Management and Object Oriented Analysis and Design. The presentation meta-model describes how graphics are represented. This description is independent of any hardware or software environment. Unlike the semantic meta-model, the presentation meta-model is not composed of multiple subject areas. The current presentation meta-model can be applied to a wide range of graphical notations.

The format of a CDIF transfer file is specified by the Transfer Format Definition. A CDIF transfer file is composed of the transfer envelope and the transfer content. The transfer envelope identifies the specific syntax and encoding used for that transfer. Currently, the Transfer Format Definition includes a single syntax and a single encoding. The syntax is an extended version of BNF (Backus Naur Form) and the encoding is a plain text representation of that syntax. The CDIF Technical Committee plans to create additional, more compact encodings to support transfer via alternate mechanisms such as inter-process communication. The transfer content conforms to the syntax and encoding specified in the transfer envelope. The transfer content is composed of the header, the meta-model section and the model section. The meta-model section identifies a version of the CDIF meta-model and includes any exporter defined extensions to that meta-model. The information in the meta-model section is used to describe the contents of the model section. The model section contains the model being transferred.

The extensibility features of the CDIF meta-model are specified by the Framework for Modeling and Extensibility. This framework provides the meta-meta-model that is used to describe the CDIF meta-model. An exporter may extend the CDIF meta-model in conformance to the CDIF meta-meta-model. The extended meta-model, however, may not conflict with the CDIF meta-model. The framework also defines the responsibilities of the importing and exporting tools when interchanging models conforming to an extended CDIF meta-model.

Relevance to Asset Interchange CDIF provides a reasonable mechanism for interchanging data models among asset libraries. The advantages to this mechanism are support for multiple subject area meta-models and existing specifications of 3 subject area meta-models. However, CDIF is only a draft EIA standard and has not yet been proposed as an ANSI or ISO standard. Also, the CDIF meta-model is extensive (and still growing) and CDIF does not allow conflicting meta-models.

CDIF does not provide any mechanisms for interchanging data among asset libraries.

7.3.3 Summary of IEEE P1175 Support for Data Interchange

The draft IEEE P1175 standard, A Standard Reference Model for Computing System Tool Interconnections [P1190], is a product of the IEEE Computer Society's Task Force on Professional Computing Tools. The draft standard is composed of reference models for: tool to organization interconnections, tool to platform interconnections and information transfer among tools. The draft standard also includes the Semantic Transfer Language (STL). This summary will address the reference model for information transfer among tools and the STL.

Approach The reference model for information transfer among tools describes the mechanisms and a process for transferring information. The transfer mechanisms considered are transfer via inter-process communication, via file, via shared data repository and via network communication. The transfer process includes a "receiver beware" policy and descriptions of the send service and the receive service needed to implement the process. The "receiver beware" policy states that the sender transmits as much information as possible and that the receiver discards any information that it does not need or understand. The reference model also identifies the need to represent the semantics of the information being transferred. The STL is intended to satisfy this need.

The design goals for the STL are as follows: the language must be parseable, it must be easy to read and write without special training or special tools and a transfer of information using the STL must make efficient use of machine resources. To meet these goals, the STL uses a syntax similar to natural language and the draft standard provides a technique for converting this syntax into a more efficient representation. The STL syntax is described using a BNF notation.

The STL information packet is the unit of transfer for tools using the STL. This packet is composed of an STL identification sentence, zero or more STL sentences and a packet end mark. The STL identification sentence contains packet identification information such as the originator of the packet and a timestamp. An STL sentence, the STL equivalent of a natural language sentence, consists of exactly one subject and one or more clauses. Each clause is composed of a verb or verb phrase and a direct object. Each clause expresses exactly one relationship to or attribute of the subject. The subject of an STL sentence is restricted to defined concept names. Likewise, each STL clause must include a defined relationship or attribute keyphrase. The draft standard includes concept names, relationship keyphrases and attribute keyphrases that can be used to support various modeling techniques such as entity-relationship modeling, data flow modeling, state transition modeling and data typing. STL users may define their own concepts and keyphrases using the STL extensibility mechanism.

Relevance to Asset Interchange IEEE P1175 provides a reasonable mechanism for interchanging data models among asset libraries. The advantages to this mechanism are a natural language-like syntax and support for multiple meta-models. One disadvantage to this approach is that the draft standard does not clearly distinguish between the supported meta-models and the syntax used to represent those meta-models. Indeed, the draft standard does not even identify which meta-models are supported. It merely provides examples of how the defined concepts and keyphrases could be used to support various meta-models.

IEEE P1175 does not provide any mechanisms for interchanging data among asset libraries.

7.3.4 Other Standards

The standards evaluation activity is intended to be a comprehensive evaluation of standards that may include data interchange capabilities applicable to the long term approach. Although the sections above summarize the most applicable standards, other standards were also evaluated and there are some standards that have yet to be evaluated.

The Portable Common Tool Environment (PCTE) [PCT90], A Tool Integration Standard (ATIS) [ATI90] and the Standard for the Exchange of Product Model Data (STEP) fall into the category of standards that have been evaluated. Summaries of PCTE and ATIS were not produced because, although these standards both include well defined meta-models and data interchange services, neither standard defines an external representation for its data. A summary of the STEP standard was not produced because, although this standard is particularly applicable to data interchange, it will not mature within the timeframe of ALOAF development. The current STEP standard is not intended to address software products. A STEP standard that will address software products will not be available for at least another two years.

The Hypermedia/Time-based Structuring Language (HyTime) [HYT90], Engineering Information Systems (EIS) [EIS86] and the products of the Computer-aided Acquisition and Logistic Support (CALS) program fall into the category of standards that have yet to be evaluated. Each of these standards will be evaluated and their relevance to the asset interchange long term approach will be determined.

7.3.5 Initial Conclusions

The first conclusion drawn from the evaluation activity is that a meta-model based approach to data interchange is the correct approach. There is ample precedence in the standards community for this conclusion. At least three standards, CDIF, ANSI IRDS and IEEE P1175, have chosen this approach and two others, PCTE and ATIS, are likely to do so. The second conclusion drawn from this activity is that there is a strong emphasis in the standards community on the interchange of data models but only a weak emphasis on the interchange of data. In the standards evaluated so far, there are three formats that can be used to represent data models but none that can be used to represent data. However, the standards evaluated represent the state of the art in data interchange. Standards with a less sophisticated view of data interchange may provide adequate formats for representing data. The final conclusion drawn from this activity is that the standards community provides a large body of work on data interchange and that this work should be reused in the development of the Asset Interchange Specification.

8 Services

This section provides the detailed definition of the ALOAF services first introduced in section 6. This edition provides a listing of some proposed services with particular emphasis on the Data Model services. In future editions, discussion within each of the service categories will be expanded to include all of the core services in each category. The services are described from the points-of-view of their interfaces, their basic behavior and possible error conditions noted during service execution. Eventually, all service categories will be developed into full language-independent service specifications culminating in a proposed set of Ada package specifications of service categories and individual services.

Asset library services exist at several levels (see figure 1 on page 2). Services apparent to a user (including library administrators, system administrators, individuals looking for assets, and others) can be provided by a combination of tools and facilities built into the library and its host operating environment. ALOAF specifies a minimal set of core services which provide access to the library and its contents. These services are intended to provide a host-independent interface to support reuse tool portability across various heterogeneous environments, with minimal constraints on the capabilities or design of any particular library implementation.

Extended services may be defined in future versions of this document for particular library products, families, domains, etc. These services would provide access to capabilities that might not be meaningful to all libraries. For example, a local project library may need more extensive provisions for security, library access, extended search, transaction logging, etc. than the core services provide.

Even more complex services, such as graphical interactive browsing assistance, will be provided by tools which use the ALOAF service interfaces defined here (as well as others available in the host environment). These tools will vary from one environment to the next and may or may not be considered part of a vendor's library product. The specification of such tools is beyond the scope of the ALOAF.

The reader should note that several areas of ALOAF services which affect library management activities have been deferred to later releases of the ALOAF. In particular, access control and transaction processing have not been well developed. As a basis for conceptualization, our working assumption is that changes to libraries, data models, etc. are "committed" (made permanent) only upon successful completion of a `Close` operation for which the corresponding `Open` was made explicitly for write access. All users are tacitly assumed to have full read/write access to all library elements. These areas will be revisited in future editions of this document.

8.1 Library Management Services

This category includes services associated with accessing a pre-existing library, and with the establishment of a new library, using the structure of a data model that has been developed using data model services.

Interfaces

Procedure:	`Open_Library`
In:	*Library_Name* : `Name_Type`
In:	*Read_Only_Flag* : `Boolean`
Out:	*Library_Id* : `Library_Id_Type`

Procedure:	`Close_Library`
In:	*Library_Id* : `Library_Id_Type`
In:	*Abort_Flag* : `Boolean`

Procedure: `Get_All_Libraries`
Out: *All_Libraries* : `Library_Set_Type`

Procedure: `Initialize_Library_Iterator`
In: *Library_Set* : `Library_Set_Type`
Out: *Library_Iterator* : `Library_Iterator_Type`
Out: *Iteration_Status* : `Iterator_Status_Type`

Procedure: `Next_Library_From_Iterator`
In Out: *Library_Iterator* : `Library_Iterator_Type`
Out: *Next_Library* : `Library_Id_Type`
Out: *Iteration_Status* : `Iterator_Status_Type`

Procedure: `Get_Library_Name`
In: *Library_Id* : `Library_Id_Type`
Out: *Library_Name* : `Name_Type`

Procedure: `Create_Library`
In: *Library_Name* : `Name_Type`
In: *Data_Model_Id* : `Model_Id_Type`
Out: *Library_Id* : `Library_Id_Type`

Procedure: `Delete_Library`
In: *Library_Id* : `Library_Id_Type`

Procedure: `Rename_Library`
In: *Library_Id* : `Library_Id_Type`
In: *New_Library_Name* : `Name_Type`

Procedure: `Get_Library_Data_Model`
In: *Library_Id* : `Library_Id_Type`
Out: *Data_Model_Id* : `Model_Id_Type`

Description

The `Open_Library` procedure requests access to a pre-existing library named *Library_Name*. The *Read_Only_Flag* provides an indication to the system that write access and consequent locking will not be required. The procedure returns a handle *Library_Id* which is used for further operations on the library.

`Close_Library` terminates access to a library previously opened and assigned the handle *Library_Id*. The *Abort_Flag* signals the system that all changes made by this user to the library since it was opened are to be ignored; otherwise, they are committed to permanent storage. (This flag has no effect if the library was opened with the *Read_Only_Flag* set.

An ALOAF library installation may consist of several libraries all accessed via an ALOAF service framework. `Get_All_Libraries` produces the set of libraries in the system and returns this set through the parameter *All_Libraries*. This set is used by the next two services, `Initialize_Library_Iterator` and `Next_Library_From_Iterator`, to provide a handle to each library in the set in turn. (For a full discussion of iterators and their use, see the Class Services subsection of the Data Model Services, which follow.)

The procedure `Get_Library_Name` returns the name of a library with a handle of *Library_Id*. The handle may have been determined previously by `Open_Library`, by `Create_Library`, or by iterating through a set returned by `Get_All_Libraries`.

`Create_Library` reserves a *Library_Name* in the system name space and returns a handle *Library_Id*. The library name must be unique within the system name space. The library thus "created" contains the structure of the data model identified by *Data_Model_Id*. (The data model is presumed to have been created prior to the library using the Data Model Services described in the next session. There is one data model per library.)

`Delete_Library` is the converse procedure. It removes the library, its data model, *and all associated asset descriptions* from the system.

`Rename_Library` provides a way to change the name of a pre-existing library.

The `Get_Library_Data_Model` procedure returns the *Data_Model_Id* of a data model associated with the input *Library_Id*.

Errors

`TBD`

8.2 Data Model Services

Assets in an ALOAF library are organized according to some conceptual scheme known as the library data model. The library's data model consists of a description of the types of information that are used to describe assets in the library. For example, if a library includes provisions for searching by facets, key words, abstracts, author, release date, and so forth, the data model for this library might provide the name of each facet, an enumeration of the allowable terms associated with each facet, the number of characters allowed in a keyword, range of integer information, format of date descriptors, and similar data-typing information. The data model also defines relationships among assets such as between code and design or between organizations and authors or implementors.

Since all data in the library must conform to the library's data model, basic changes to the data model have the potential to affect the entire library. Thus once established, the basic core of a library data model is expected to change infrequently.

Data Model services are provided to manipulate library data models in accordance with the ALOAF meta-model (see section 7.1). The services presented in this section will be modified accordingly as the ALOAF meta-model evolves.

In the service descriptions that follow, data model services are further broken down into the sub-categories:

- model definition services
- object class services
- attribute services
- relationship services

to coordinate the description of these services to the primary aspect of the meta-model that they address.

The data model services are analogous to the commands to create and alter tables in a relational database management system (RDBMS). That is, they deal with the basic structure of library data, rather than with the data itself. This kind of activity is relatively infrequent, and carries with it inherent hazards and restrictions. For example, deleting attributes or reducing the allowable range of a numeric attribute could have global effects on data already stored—such actions may require a major maintenance activity, such as regeneration of all or part of the library database. (For libraries providing continuous 24-hour-a-day service, some means must be provided to commit to the new model instantaneously.) On the other hand, adding attributes or increasing a numeric range may not have such drastic implications. Because of the potential widespread effects resulting from their use, many data model services and the tools that invoke them are most likely restricted to system or library administrators.

Asset libraries are essentially databases whose elements are "software assets" and descriptive information. As with databases in general, the ways they can be implemented are limitless. For example, a rudimentary library could be built in such a way that its "catalog" of descriptive information is stored as formatted records in one or more library files. The data model for this library would be embodied in the record structure definition statements in the source code of the library tools themselves. Changing the data model for this library would require modifying those record structure definitions and recompiling the library tools (plus a one-time operation to translate the previous catalog to the new format). A more flexible library might implement its catalog in some DBMS, and store its assets in a formal configuration management system like SCCS or CMS. Or, some or all of the assets might be kept offline on tape or (in the case of documentation) on paper. A future library design might consist of entries in a large-scale object management system, tightly integrated with development tools, where assets may not even exist as unique distinguishable files and the distinction between "assets" and "asset descriptions" becomes blurred.

In an integrated environment, the library data model could possibly be just one part of a system-wide data model. While in some implementations, the ALOAF data model services may provide access to the larger system-wide object manager, such implementation alternatives are not specifically addressed by the ALOAF. On the other hand, the concept of the library data model is broad enough to include information other than asset-specific information. Examples of such information that a library might wish to store could include: a count of library accesses; the definition of a local accounting form; metrics which store search success rate or library revenues, and so forth. The ALOAF service implementations should not artificially restrict the ability to define (and instantiate and access) such information.

Libraries which support continuous service will have a close tie between data model services and asset location services, session services, and underlying SEE transaction and locking services to allow a data model modification to be committed with minimal interruption. Editing, storage, rename, etc. of the data model is closely related with the same operations on the asset descriptions (see the Asset Description Services below).

In general, unless otherwise specified, all input parameters supplied to services are assumed to be valid instances of the corresponding type of the parameter.

8.2.1 Model Definition Services

Interfaces

Procedure:	`Create_Data_Model`
In:	*Model_Name* : `Name_Type`
Out:	*Model_Id* : `Model_Id_Type`

Procedure:	`Open_Data_Model`
In:	*Model_Name* : `Name_Type`
Out:	*Model_Id* : `Model_Id_Type`

Procedure:	`Close_Data_Model`
In:	*Model_Name* : `Name_Type`
In:	*Model_Id* : `Model_Id_Type`

Procedure:	`Delete_Data_Model`
In:	*Model_Id* : `Model_Id_Type`

Description

The `Create_Data_Model` procedure returns a handle to a newly created empty data model through the parameter *Model_Id* whose name is given by *Model_Name*.

The `Open_Data_Model` procedure returns a handle *Model_Id* to a data model in memory retrieved by name *Model_Name* from the persistent store of available library data models.

`Close_Data_Model` updates the model named *Model_Name* in persistent store with the model obtained from the handle *Model_Id*.

Given a handle *Model_Id* to a data model, `Delete_Data_Model` will remove that model from memory.

Errors

The procedure `Create_Data_Model` must note the error condition **Model_Already_Exists** if a model with the name *Model_Name* already exists within persistent store.

If there is no data model with name *Model_Name*, the procedure `Open_Data_Model` must note the error condition **No_Model**. In this case, the handle *Model_Id* will be null.

`Close_Data_Model` always succeeds. If no model with the name *Model_Name* already exists, it is written to persistent store for the first time; otherwise the model with this name is overwritten with the new model referenced by *Model_Id*.

The procedure `Delete_Data_Model` always succeeds.

8.2.2 Object Class Services

Interfaces

Procedure:	`Create_Class`
In:	*New_Class_Name* : `Name_Type`
Out:	*New_Class* : `Class_Id_Type`

Procedure:	`Delete_Class`
In:	*Class_Id* : `Class_Id_Type`

Procedure:	`Get_Class_Name`
In:	*Class_Id* : `Class_Id_Type`
Out:	*Class_Name* : `Name_Type`

Procedure:	`Rename_Class`
In:	*Current_Class* : `Class_Id_Type`
In:	*New_Class_Name* : `Name_Type`

Procedure:	`Get_All_Classes`
In:	*Model_Id* : `Model_Id_Type`
Out:	*All_Classes* : `Class_Set_Type`

Procedure:	`Get_Child_Classes`
In:	*Model_Id* : `Model_Id_Type`
In:	*Current_Class* : `Class_Id_Type`
Out:	*Child_Classes* : `Class_Set_Type`

Procedure:	`Get_All_Subclasses`
In:	*Model_Id* : `Model_Id_Type`
In:	*Current_Class* : `Class_Id_Type`
Out:	*All_Sub_Classes* : `Class_Set_Type`

Procedure:	`Get_Parent_Class`
In:	*Model_Id* : `Model_Id_Type`
In:	*Current_Class* : `Class_Id_Type`
Out:	*Parent_Class* : `Class_Id_Type`

Procedure:	`Get_All_Superclasses`
In:	*Model_Id* : `Model_Id_Type`
In:	*Current_Class* : `Class_Id_Type`
Out:	*All_Super_Classes* : `Class_Set_Type`

Procedure:	`Initialize_Class_Iterator`
In:	*Class_Set* : `Class_Set_Type`
Out:	*Class_Iterator* : `Class_Iterator_Type`
Out:	*Iteration_Status* : `Iterator_Status_Type`

Procedure:	`Next_Class_From_Iterator`
In Out:	*Class_Iterator* : `Class_Iterator_Type`
Out:	*Next_Class* : `Class_Id_Type`
Out:	*Iteration_Status* : `Iterator_Status_Type`

Procedure:	`Next_Class_By_Name_From_Iterator`
In Out:	*Class_Iterator* : `Class_Iterator_Type`
In:	*Class_Name* : `Name_Type`
Out:	*Next_Class* : `Class_Id_Type`
Out:	*Iteration_Status* : `Iterator_Status_Type`

Procedure:	`Add_Class`
In:	*Model_Id* : `Model_Id_Type`
In:	*Parent_Class* : `Class_Id_Type`
In:	*Child_Classes* : `Class_Set_Type`
In:	*New_Class_Name* : `Name_Type`
In:	*New_Class* : `Class_Id_Type`
Out:	*Model_Update_Status* : `Model_Status_Type`

Procedure:	`Remove_Class`
In:	*Model_Id* : `Model_Id_Type`
In:	*Parent_Class* : `Class_Id_Type`
In:	*Child_Class_Name* : `Name_Type`
Out:	*Removed_Class* : `Class_Id_Type`
Out:	*Model_Update_Status* : `Model_Status_Type`

Description

The `Create_Class` procedure returns a handle through the parameter *New_Class* to a newly created class whose name is given by the parameter *New_Class_Name*. This class is not associated with any data model. Classes must be created before they can be used with a data model. `Delete_Class` removes all memory associated with a class. Classes should be destroyed after they have been removed from a data model.

Given the class handle *Class_Id*, the procedure `Get_Class_Name` returns the name of the class via the parameter *Class_Name*. The procedure `Rename_Class` renames the class identified by *Class_Id* with the name *New_Class_Name*.

Get_All_Classes produces the set of classes in the data model identified by *Model_Id* and returns this set through the parameter *All_Classes*.

Get_Child_Classes returns the set of all classes that are direct descendents of the class identified by the parameter *Current_Class* within the data model identified by *Model_Id*. The set is returned through the parameter *Child_Classes*. The procedure **Get_All_Subclasses** is the logical generalization of **Get_Child_Classes** wherein all descendents, whether they are direct or indirect, of the class identified by *Current_Class* are returned through *All_Sub_Classes*.

The procedure **Get_Parent_Class** and its generalization **Get_All_Superclasses** are inverses to the preceding two services in that they return a class or set of classes that are ancestors to the class identified by *Current_Class*. The first one collects only the immediate the parent of this class, while the second one retrieves all ancestor classes, both direct and indirect, through the parameters *Parent_Class* and *All_Super_Classes* respectively.

The next three services provide the means to process sets of classes through the use of explicit iterators. Through iterators, it is possible to process all members of an unordered collection sequentially, obtaining each set member exactly once. First an iterator must be initialized (**Initialize_Class_Iterator**) so that it is able to process the set from some logical starting position. *Class_Set* provides the set to be iterated over; the two output parameters return the iterator structure *Class_Iterator* (to be used by the other iteration services), and a status variable *Iteration_Status* to monitor the progress of the iteration. The latter parameter is discussed more fully in the Errors paragraph given below. The next two services enable successive processing of members of a set of classes. Each of them maintains the current state of the iterator via *Class_Iterator*. **Next_Class_From_Iterator** gets the next class from the set and returns it through *Next_Class* and notes the iteration status. **Next_Class_By_Name_From_Iterator** will advance the iterator until a set element is found with the name supplied through *Class_Name*, or until the end of the set is reached. *Iteration_Status* is updated as necessary.

The next two services address the processing of classes in the context of a given data model which is supplied through the parameter *Model_Id*. **Add_Class** identifies specific classes in the model as the parent and child classes for a new class *New_Class* to be placed within the model. If there are no errors (cf. the Errors section below) in these parameters, the model is updated by the insertion of the new class. The status parameter *Model_Update_Status* will indicate a successful model update or note one of a number of error conditions. **Remove_Class** is the inverse operation to the addition of a class. It is restricted, however, in that only "leaf" classes can be removed. That is, a class to be removed can have no subclasses of any kind. All links from the class *Parent_Class* to the removed class are broken. The indicator parameter *Model_Update_Status* will identify any problems noted with the removal.

Errors

Service**Create_Class** must note the error **No_Storage_Found** if there is insufficient storage to hold a new class. **Delete_Class** must note the error **No_Storage_Freed** if the storage occupied by the class *Class_Id* could not be released.

Services **Get_Class_Name**, **Rename_Class** and **Get_All_Classes** always succeed, assuming that the input parameters are valid. **Get_All_Classes** will return a null set if the model has no classes.

The four services obtaining sub- or superclasses of a given class must note the error **Not_In_Model** if the specified class is not contained in the data model accessed by *Model_Id*. Otherwise these services will always succeed, though they may produce a null set.

The iterator services all depend on a particular iterator storage structure. Any problems within these services in accessing or initializing this structure must note the condition **Invalid_Iterator**. In addition, certain noteworthy status indications about iterators should be provided through *Iteration_Status*. `Initialize_Class_Iterator` will set *Iteration_Status* to **Set_Empty** if the iterator structure was successfully initialized but the set to be iterated over is empty. Otherwise, successful initialization will set *Iteration_Status* to **First_Element**. Executions of either of the iterator forms producing the next element in a set must set the parameter *Iteration_Status* to **Set_Empty** when the end of the set is reached. In this case, the output parameter *Next_Class* will be the null set. Otherwise, *Iteration_Status* must be set to **Next_Element** and *Next_Class* will reference this next element.

The parameter *Model_Update_Status* is used to record the the status of various operations that potentially cause changes to the model. In general, it is assumed that the parameters to these operations are meaningful; e.g., class sets are not empty and class parameters denote legal classes. For example, successful completion of `Add_Class` will set *Model_Update_Status* to **Class_Added** and successful completion of `Remove_Class` will set the parameter to **Class_Removed**. Conversely, unsuccessful executions of these services will set *Model_Update_Status* to **Class_Not_Added** and **Class_Not_Removed** respectively. `Add_Class` will fail if there already is a class with name *New_Class_Name* as a child of the class *Parent_Class* (or a class with that name which is a parent of any members of the set *Child_Classes*). The service `Remove_Class` will fail if there is no class with name *Child_Class_Name* as a child of the class *Parent_Class* or if the class identified by *Child_Class_Name* has any subclasses itself, or any asset descriptions which are identified with this class in the model. Thus a class to be removed must be essentially empty and at the frontier of the data model. In the case of unsuccessful operations, the model is not updated so that the outgoing parameter *New_Model_Id* will be identical to *Model_Id*.

8.2.3 Attribute Services

Interfaces

Procedure:	`Create_Attribute`
In:	*Attr_Name* : `Name_Type`
In:	*Attr_Spec* : `Attr_Spec_Type`
Out:	*Attr_Id* : `Attr_Id_Type`

Procedure:	`Get_Attribute_Name`
In:	*Attr_Id* : `Attr_Id_Type`
Out:	*Attr_Name* : `Name_Type`

Procedure:	`Get_Attribute_Spec`
In:	*Attr_Id* : `Attr_Id_Type`
Out:	*Attr_Spec* : `Attr_Spec_Type`

Procedure:	`Get_All_Class_Attributes`
In:	*Class_Id* : `Model_Id_Type`
Out:	*All_Attrs* : `Attr_Set_Type`

Procedure:	`Get_All_Local_Attributes`
In:	*Class_Id* : `Model_Id_Type`
Out:	*Local_Attrs* : `Attr_Set_Type`

Procedure:	`Get_All_Inherited_Attributes`
In:	*Class_Id* : `Model_Id_Type`
Out:	*Inherited_Attrs* : `Attr_Set_Type`

Procedure:	`Add_Attribute`
In:	*Class_Id* : `Class_Id_Type`
In:	*New_Attr_Id* : `Attr_Id_Type`
Out:	*New_Class_Id* : `Class_Id_Type`
Out:	*Class_Update_Status* : `Class_Status_Type`

Procedure:	`Remove_Attribute`
In:	*Class_Id* : `Class_Id_Type`
In:	*Attr_To_Remove* : `Attr_Id_Type`
Out:	*New_Class_Id* : `Class_Id_Type`
Out:	*Class_Update_Status* : `Class_Status_Type`

Procedure:	`Get_Classes_With_Attribute`
In:	*Model_Id* : `Model_Id_Type`
In:	*Attr_Id* : `Attr_Id_Type`
Out:	*Classes_With_Attr* : `Class_Set_Type`

Description

TBD

Errors

TBD

8.2.4 Relationship Services

TBD

8.3 Asset Description Services

Asset descriptions consist of class instances, known as *objects*, and relationship instances, known as *links*. The asset description services described below focus on managing and manipulating objects; the link-related services are yet to be defined.

When creating a new asset description it must be determined which objects must be created to describe the asset and, for each object, determine which class of the classification scheme is the best fit for the object being classified. Each object is instantiated from the appropriate class, called the associated class, which defines a set of attributes for which data values can be supplied. Each attribute is stored using a unique attribute identifier and it is paired with an appropriate type of value. The set of attribute/value tuples effectively determines characteristics of the object. After all the correct data values have been supplied, each asset description object can be catalogued, making them available for browsing and searching.

Asset description services are provided to create, delete, inspect, and update objects and links. As stated earlier, the data model services are analogous to the commands to create and alter tables in a relational database management system. That is, they deal with the basic structure of library data, rather than with the data itself. The Asset Description Services deal with data values and are used to create, delete, modify, browse, and retrieve the asset descriptions stored in the library. Asset description services are similar to adding, deleting, and modifying the rows of a table in a relation database management system.

An asset description object is created with an associated class that defines the set of attribute identifiers that comprise the object. The associated class contains a set of attribute identifiers that have been defined or inherited from a superclass. The object contains a set of attribute/values through which the value of each attribute identifier for the object is accessed. For each attribute identifier there may be an associated value. The values that are stored for each attribute of the asset description must match the type specified by the corresponding attribute definition.

Example: The Common Data Model defines a class called Person with attributes Name, Address, Telephone_Number, Electronic_Mail_Address, and Is_Contact_For. When an object of class Person is created, the new object contains a set of attribute/values tuples that contains an entry for each attribute defined by Person and its superclasses. The attribute values of the object can be accessed by the object identifier and the attribute identifier. Accessing the Name of the Person is performed by specifying the object identifier and the attribute identifier for the Name attribute. The string value of the name attribute is returned.

8.3.1 Object Services

Interfaces

Procedure:	`Create_Object`
In:	*Associated_Class* : `Class_Id_Type`
In:	*Class_Name* : `String`
Out:	*New_Object* : `Object_Id_Type`

Out:	*Object_Update_Status* : `Object_Status_Type`
Procedure:	`Destroy_Object`
In:	*Object_Id* : `Object_Id_Type`
Out:	*Object_Update_Status* : `Object_Status_Type`
Procedure:	`Get_Attributes_For_Asset`
In:	*Object_Id* : `Object_Id_Type`
Out:	*Attributes_Id_List* : `Attr_List_Type`
Out:	*Object_Update_Status* : `Object_Status_Type`
Procedure:	`Get_Attribute_Boolean_Value`
In:	*Object_Id* : `Object_Id_Type`
In:	*Attribute_Id* : `Attr_Id_Type`
Out:	*Attr_Value* : `Boolean`
Out:	*Object_Update_Status* : `Object_Status_Type`
Procedure:	`Get_Attribute_Integer_Value`
In:	*Object_Id* : `Object_Id_Type`
In:	*Attribute_Id* : `Attr_Id_Type`
Out:	*Attr_Value* : `Integer`
Out:	*Object_Update_Status* : `Object_Status_Type`
Procedure:	`Get_Attribute_String_Value`
In:	*Object_Id* : `Object_Id_Type`
In:	*Attribute_Id* : `Attr_Id_Type`
Out:	*Attr_Value* : `String`
Out:	*Object_Update_Status* : `Object_Status_Type`
Procedure:	`Get_Attribute_Float_Value`
In:	*Object_Id* : `Object_Id_Type`
In:	*Attribute_Id* : `Attr_Id_Type`
Out:	*Attr_Value* : `Float`
Out:	*Object_Update_Status* : `Object_Status_Type`
Procedure:	`Get_Attribute_Time_Value`
In:	*Object_Id* : `Object_Id_Type`
In:	*Attribute_Id* : `Attr_Id_Type`
Out:	*Attr_Value* : `Time`
Out:	*Object_Update_Status* : `Object_Status_Type`
Procedure:	`Set_Attribute_Boolean_Value`
In:	*Object_Id* : `Object_Id_Type`
In:	*Attribute_Id* : `Attr_Id_Type`
In:	*Attr_Value* : `Boolean`
Out:	*Object_Update_Status* : `Object_Status_Type`

Procedure:	Set_Attribute_Integer_Value
In:	*Object_Id* : Object_Id_Type
In:	*Attribute_Id* : Attr_Id_Type
In:	*Attr_Value* : Integer
Out:	*Object_Update_Status* : Object_Status_Type

Procedure:	Set_Attribute_String_Value
In:	*Object_Id* : Object_Id_Type
In:	*Attribute_Id* : Attr_Id_Type
In:	*Attr_Value* : String
Out:	*Object_Update_Status* : Object_Status_Type

Procedure:	Set_Attribute_Time_Value
In:	*Object_Id* : Object_Id_Type
In:	*Attribute_Id* : Attr_Id_Type
In:	*Attr_Value* : Time
Out:	*Object_Update_Status* : Object_Status_Type

Procedure:	Set_Attribute_Float_Value
In:	*Object_Id* : Object_Id_Type
In:	*Attribute_Id* : Attr_Id_Type
In:	*Attr_Value* : Float
Out:	*Object_Update_Status* : Object_Status_Type

Procedure:	Get_Attribute_Values_For_Object
In:	*Object_Id* : Object_Id_Type
Out:	*Object_Data* : Object_Definition
Out:	*Object_Update_Status* : Object_Status_Type

Procedure:	Get_Associated_Class
In:	*Object_Id* : Object_Id_Type
Out:	*Associated_Class* : Class_Id_Type
Out:	*Object_Update_Status* : Object_Status_Type

Description

In this section, descriptions are given to explain the important features of each service. Unless otherwise specified, all input parameters supplied to services are assumed to be valid instances of the corresponding type of the parameter.

The Create_Object procedure returns a handle to a newly created object through the *New_Object* parameter. This handle references a set of attribute/value tuples composing the the object. Each tuple corresponds to one attribute that is inherited or defined by the *Associated_Class*, Initially, blank or default values are entered for each value in the set. The *Object_Name* specifies the name for the object. This name must be unique within the all objects that are associated with the *Associate_Class*. The success of the procedure is returned in *Object_Update_Status*.

The `Destroy_Object` procedure removes the object from the library and removes all memory associated with the object. The success of the procedure is returned in *Object_Update_Status*.

The `Get_Attributes_For_Object` procedure returns the set of attributes from the attribute/value set. This set is identical to the list of attributes defined and inherited for the associated class of *Object_Id*.

Each object contains a set of attribute/value tuples that describe the object. The values of the attributes can be different types. There is a service provided to get and set attribute values for each of the different types of attributes. For example, the `Get_Attribute_Integer_Value` procedure returns in *Attr_Value* the integer value of the *Attribute* for the *Object_Id* while the `Set_Attribute_Boolean_Value` procedure sets the Boolean value of the *Attribute_Id* to *Attr_Value* for the *Object_Id*. The success of each of these services is returned in *Object_Update_Status*. Each of the other `Get_` and `Set_` services perform similarly.

The `Get_Attribute_Values_For_Object` procedure returns a list of all the attribute/value tuples for an object. This service is a convenience to be used when all attributes/values for an object are required, This service request replaces looping over all the attributes and submitting a request for the value of each attribute. The success of the procedure is returned in *Object_Update_Status*.

The `Get_Associated_Class` procedure returns the class definition that is associated with the object. The associated class is set when the object is create.

Errors

Services `Create_Object` must note the error **No_Storage_Found** if there is insufficient storage to hold a new object. If the *Associated_Class* is not a valid class identifier the error **Associated_Class_Does_Not_Exist** is returned and the object is not created. If the *Object_Name* is not unique for all object associated with *Associated_Class* the then **Name_Not_Unique** is returned.

`Destroy_Object` must note the error **No_Storage_Freed** if the storage occupied by the class *Object_Id* could not be released. `Destroy_Object` returns the error **Object_Does_Not_Exist** if *Object_Id* is not a handle of a valid object.

`Get_Attributes_For_Object` notes the error **Object_Does_Not_Exist** if the handle of *Object_Id* is not the handle of a valid object.

There is a service provided to get and set the values for each different type of attribute. Each of these services return the same error messages. If the *Object_Id* is not valid the error **Object_Does_Not_Exist** is returned. When the *Attribute_Id* is not valid the error **Attribute_Does_Not_Exist** is returned.

When the *Object_Id* is not valid the service `Get_Attribute_Values_For_Object` returns the error **Object_Does_Not_Exist**.

8.3.2 Link Services

`TBD`

8.4 Session Services

This section describes how to initiate access to asset libraries. All library interaction is within the context of a session, which establishes the resources available for use by a client application. (It is presumed for this discussion that a "client application" and its resources are associated with a single user operating within the context of a single operating system process.) When a session is terminated, all resources are returned. An application can have only one session open at a time (however, some library implementations may have sessions active simultaneously for different users).

Interface

Procedure:	`Open_Session`
In:	*User_Id* : `User_Id_Type`
Out:	*Session_Id* : `Session_Id_Type`
Procedure:	`Close_Session`
In:	*Session_Id* : `Session_Id_Type`

Description

The `Open_Session` procedure establishes a session and returns a handle *Session_Id* which is a unique identifier for the session. All subsequent library access is done in the context of this session. The *User_Id* is provided to the user by prior arrangement with the librarian or system administrator; it may be used in conjunction with library-specific access controls beyond those available via the host SEE security provisions. A null value may be supplied; this provides whatever access is associated with the user's system login or a system default for public access. Conformant implementations may require that any existing session be terminated before a new one is initialized.

The `Close_Session` procedure ends a session and frees resources associated with it. Element handles returned during the session are no longer valid.

Errors

The procedure `Open_Session` may return the condition **Invalid_User_Id** if an unknown *User_Id* was supplied or if it was null and the library provides no default access. `Open_Session` returns the condition **Session_Not_Opened** for any other error in attempting to start a session. Any error in initiating a session returns a null *Session_Id*; the session is not started and no resources are allocated.

The procedure `Close_Session` returns the condition **Session_Not_Closed** for any error encountered during session termination.

8.5 Query Services

The query services provide a means to search a library's data model (and consequently the asset descriptions) in accordance with a set of criteria. One problem to be solved here is that the tool which transmits the search criteria (an interactive browser, for example) will generally have little a priori knowledge of the structure of the data model used by the library being searched, while truly seamless operation requires as little human intervention as possible to discover that structure. However, search tools can take advantage of the universal meta-model upon which ALOAF libraries are constructed.

The response to a query is a list of class and/or object id's which can be accessed one at a time and used in further queries or in a request to view (parts of) an asset. Queries may be iterative, with a library user supplying successively stronger criteria to refine a search until a set of assets of interest is identified.

One possible approach to query services is to provide a query language specification along with a general query language processing service. Such a service must be able to produce sets of class and/or object id's which have the properties determined by a given query language expression. With this approach, the basic decisions that must be made in defining the query services deal with the syntax and semantics of the query language.

In this context, various kinds of queries can be performed. One is a boolean query. Another query is an information retrieval query where words or phrases are searched for in text attributes. These information retrieval searches can be enhanced by extending the search to related terms, synonyms, broader terms, and narrower terms. The selected query language(s) should allow the user to be able to specify any of these searches.

Another approach is to provide a standard set of information retrieval services. Such operations can be broadly defined as:

- get class ids
- get object ids

These operations return a list of class or object id's in response to a generalized query that includes classification terms, text-in-context and other attributes, specified as regular expressions. The various forms of the get operation are expressed in terms of query qualification parameters which set bounds for the search for relevant classes or objects. The form of such query "templates" is highly dependent on a library's meta-model. The semantics (or interpretation) of the results of a query are in turn dependent on a library's data model.

A specific approach to the ALOAF query services has not yet been chosen. The specific services and associated query language(s) will be specified in a future edition of the ALOAF.

8.6 Asset Location Services

Asset location services enable the examination and, consequently, transfer of assets from a library to a user or tool. Services here must deal with what and how much may be shown/transferred to whom and what side effects may occur as a result. For example, transfer of an asset may occasion some check for user status (security check, fees paid, release waiver acknowledged, etc), may imply a search through offline storage and physical handling/mailing of alternate media, and may trigger accounting activities to support such things as notification of upgrade availability or product recalls. Note that such events as these will be handled by tools built using ALOAF services and not by ALOAF functions themselves. This implies that different parts of an asset could have different relative values—for example, a commercial library service may make text descriptions and abstracts available to anyone for "free," but source code, design description documents, and user's manuals would require payment of a fee, assigned by the library administration.

Some general kinds of services in this category include:

- locate asset
- send asset request

Locating an asset means that its system 'address' (or the equivalent if the actual asset is not available electronically) is provided to allow the user or tool to access an asset. In the case of assets that are not directly available, asset request services provide the means to eventually obtain an "off-line" asset.

The ALOAF does not restrict libraries to specific styles and methods of providing access to assets themselves. It may be the case that no assets are made available directly—all asset requests require some off-line action to take place. Conversely, a library may contain all of its assets in an on-line archive.

8.7 Metrics Services

Services which control the collection and disposition of statistical information are in this category. Libraries will generally be expected to collect information on assets and their usage and to make some portion of the information available to users and administrators.

Metrics services include:

- request metrics list
- read metric value
- store metric value

Requesting a metrics list will return a list of available metrics associated with some current user context. Thus metrics list services will require object id's and user id's as input parameters. Once a metric item has been found, its value can be provided. Users probably cannot store metric values directly, but the tools they use may do so—e.g., locating an asset may increment an access 'hit' counter which is maintained by the library.

8.8 Access Control Services

Access control services help maintain the wide variety of restrictions that may be placed on assets and on users either legally or by policy. Primary reliance will be placed on the access control services provided by the underlying SEE. However, library assets demand some additional consideration arising from the varied sources of assets and their relatively long persistence. In general, library assets will come encumbered with a variety of restrictions reflecting the ownership rights of the originating organization and, potentially, each subsequent modifying organization. The encumbrances may not be homogeneous—an asset may be free to one organization (Company X, who has paid for unlimited rights to updates) or class of organization (educational institutes) and not available to yet another class (nonpaying browsers, company competitors, etc.); some parts of assets will likely be open to anyone (text abstracts) and others variously restricted (source code, users' manuals). Libraries must also be able to maintain and enforce restrictions on imported assets and to forward the same restrictions (perhaps with others added) when assets are exported.

Some sample access control services include:

- create acl
- assign entity to acl
- delete acl

Note that the preceding list of services suggests the use of Access Control Lists (ACLs) to control library access. This approach may be replaced by a more general approach in the future. It is likely that an access control policy will be part of a library's data model, so that general access control service descriptions will be provided in terms of the ALOAF meta-model.

9 Conformance

The ALOAF defines a collection of services and specifications that not all library systems may be able to fully support. There are likely to be a variety of ways in which a library system can conform to the ALOAF. While we have not yet fully analyzed this issue, we have reached some preliminary conclusions.

Conformance can be considered from the viewpoint of the *quantity* of services provided, the *quality* of services provided, and the *mode* in which services are provided. Each of these views can be considered a different conformance dimension. Within each of these dimensions, there are likely to be several different classes of conformance (which may be viewed as levels of conformance if there are hierarchical gradations between classes). Although the conformance dimensions are not strictly independent, they provide a convenient framework for discussing conformance issues.

9.1 Conformance Dimensions

Service Quantity

This dimension addresses the amount of ALOAF services a particular library system supports. A minimal conformance level within this dimension may be one in which no ALOAF-conformant programmatic services (per section 8) are available, but asset import and export capabilities in accordance with the Asset Interchange Specification in section 7 are supported. Beyond this minimal level of conformance, the Service Quantity dimension will likely be shaped by specifying both a set of core ALOAF programmatic services and a set of extensions to those core services. Every library system purporting to provide ALOAF-conformant programmatic services must then provide at least the core services, and the Service Quantity of such systems will then be characterized by which of the extensions (if any) they provide.

In future versions of this document, the Services section will identify both core and extended services, and the extended services will be classified appropriately to define specific conformance classes. It is possible that entire service categories will be considered extended services, but it is more likely that particular collections of services within (and possibly across) the categories will be identified as extensions. For example, if the ALOAF meta-model has core and extended portions, the extended portions will be reflected in a set of extended services within at least the data model and asset description service categories. Other potential candidates for extended services may include metrics services, access control services, and some library management, session, query, and import/export services.

Service Quality

This dimension exists in recognition that different library systems may provide some services with varying degrees of quality or engineering sophistication. The focus in this dimension is on issues that are related to what are typically referred to as non-functional requirements. These can include time and space efficiency, assurance and reliability, availability, and environmental context, among others. However, the emphasis will be on factors that have practical impact on a library system's ability to meet functional ALOAF requirements.

Some specific library system issues that may be addressed by this dimension include:

- the communications, processing, and storage performance factors affecting the system's ability to provide remote sites with direct on-line access to programmatic services, and thus affecting the Service Quantity that the system can practically deliver (not just make available) to remote clients,
- the overall assurance level of the system and its underlying operating system or software engineering environment in support of data security/integrity services and general system reliability, and
- the characteristics of the system and its underlying environment that impact library capacity, library availability, error recovery, connectivity with other libraries, etc.

Service Mode

This dimension addresses the interface or service modes in which a library system presents services to client applications. The two major interface modes that the ALOAF may eventually address are procedural and protocol modes. A library system supporting a procedural interface mode provides services through language (e.g., Ada) bindings requiring the client application to be linked and loaded with the system software at run time. A library system supporting a protocol interface mode provides services via a message-passing communications protocol enabling an asynchronous client-server relationship between the application and the system. The protocol interface mode is considered the more general of the two; client applications in this mode can interact directly with the server via the communications protocol, but they more typically interact via a procedural binding to communications services supporting the protocol that simulates a direct procedural interface from the application's viewpoint. The ALOAF will initially support the procedural interface mode by defining an Ada programmatic interface to ALOAF services, but a longer term goal is to define an underlying protocol interface to more directly address the long-term ALOAF goal to support seamless interoperability of distributed, heterogeneous libraries.

Note that this dimension may be related to the Service Quantity dimension in that differing quantities of services may be provided by a library system in the two interface modes. It may also impact the Service Quality dimension from a performance perspective.

9.2 ALOAF Conformance Plans

Future versions of this document will more clearly define the conformance dimensions, define specific conformance classes within those dimensions (e.g., by defining a set of extended services and mapping them to Service Quantity conformance classes), and define criteria for determining conformance in each class. It might prove useful to define an alternative view of the conformance classification scheme by aggregating conformance classes in various dimensions into a set of overall ALOAF conformance classes (for example, a library system exhibiting Service Quantity A, Service Quality B, and Service Mode C might be said to be in overall ALOAF conformance class X).

However, the intent of this effort will be strictly to establish a useful classification scheme for libraries with respect to their ability to provide ALOAF capabilities. It is beyond the scope of the current ALOAF effort to provide methods and tools to assess, validate, or enforce the conformance of any particular library.

10 References

[ATI90] Specification for A Tool Integration Service (ATIS). CASE Integration Services Committee, November 1990. CIS Base Document V1.0.

[Boo91] Grady Booch. *Object-Oriented Design with Applications.* Benjamin/Cummings, 1991.

[CD90] Edward R. Comer and Cameron M. Donaldson. Product Definition Document for the Automated Reusable Components System (ARCS). Software Productivity Solutions, Indialantic, FL, October 1990. Volume 1: System Description.

[CDI91a] CDIF - Framework for Modeling and Extensibility. Electronic Industries Association (EIA) CASE Data Interchange Format (CDIF) Technical Committee, April 1991. Draft Interim Standard Version 1.40 EIA-PN2387.

[CDI91b] CDIF - Standardized CASE Interchange Meta-Model. Electronic Industries Association (EIA) CASE Data Interchange Format (CDIF) Technical Committee, April 1991. Draft Interim Standard Version 1.40 EIA-PN2329.

[CDI91c] CDIF - Transfer Format Definition. Electronic Industries Association (EIA) CASE Data Interchange Format (CDIF) Technical Committee, April 1991. Draft Interim Standard Version 1.40 EIA-PN2389.

[Ear90] Anthony Earl. A Reference Model for Computer Assisted Software Engineering Environment Frameworks, August 1990. Version 4.0 ECMA/TC33/TGRM/90/016.

[EIS86] Requirements for Engineering Information Systems. Institute for Defense Analyses, Alexandria, VA, July 1986.

[HYT90] Hypermedia/Time-based Structuring Language (HyTime), September 1990. ANSI X3V1.8M/SD-7 (HyTime) Seventh Draft.

[IRD88] Information Resource Dictionary System (IRDS). American National Standards Institute (ANSI), October 1988. ANSI X3.138-1988.

[P1190] A Standard Reference Model for Computing System Tool Interconnections. IEEE Computer Society Task Force on Professional Computing Tools, October 1990. Draft P1175/D7.

[PCT90] Portable Common Tool Environment (PCTE) Abstract Specification. European Computer Manufacturers Association (ECMA), November 1990. Final Draft ECMA/TC33/90/78.

[RIG91] Charter of the Reuse Library Interoperability Group, May 1991.

[STA91a] STARS Reuse Concept of Operation, August 1991. Version 0.5, Unisys CDRL Sequence No. 03725/001/00, Publication Number GR-7670-1251.

[STA91b] STARS Vision, May 1991. Version 0.1.

[SWT89] J. Solderitsch, K. Wallnau, and J. Thalhamer. Constructing Domain-Specific Ada Reuse Libraries. In *Proceedings, 7th Annual National Conference on Ada Technology*, March 1989. pages 419-433.

Part IV

Appendices

A Glossary

ASSET Any unit of information of current or future value to a software-intensive systems development and/or PDSS enterprise. Assets may be characterized in many ways including as software-related work products, software subsystems, software components, contact lists for experts, architectures, domain analyses, designs, documents, case studies, lessons learned, research results, seminal software engineering concepts and presentations, etc.

ASSET CATALOG The collection of asset descriptions which an asset library maintains about its assets, as well as the data model by which the library is organized.

ASSET DESCRIPTION The information about an asset that is kept by the library in the context of the library's data model. This is generally the information provided to a library user in response to a query. It is the information made available to assess the suitability of an asset to fulfill whatever purpose the requestor has in mind.

ASSET INTERCHANGE The act of transferring one or more assets from one asset library to another. The STARS view is that this includes the capability to transfer asset descriptions in a library-independent representation. The data model that describes this representation may be understood, as in the case of the Common Data Model, or may be explicitly transferred as part of an asset interchange.

> **COMMON DATA MODEL** A representation of information that is commonly maintained by some asset libraries using the STARS library mechanisms. The Common Data Model supports a rudimentary asset interchange capability which is the basis for the STARS short-term interchange approach.
>
> **DATA FORMAT** The Asset Interchange Data Format is an asset library-independent representation of library data models and data. It provides a long-term, meta-model-based approach to library asset interchange.

ASSET LIBRARY A collection of software assets controlled by an asset library system. Typically, asset libraries are implemented using an asset library system, which is a computer-based system designed to facilitate the reuse and sharing of software assets. Asset libraries provide a set of services that support qualifying, reusing, and managing software assets.

> **AUTOMATED REUSE ASSET LIBRARY** An asset library with automated tools and services that facilitate operations such as search/query/browse, asset interchange, and interoperation with other libraries and with reuse tools.
>
> **DISTRIBUTED ASSET LIBRARY** (1) A single library with an on-line database of asset descriptions and/or assets residing on two or more physically distinct computers. (2) A logically single library which consists of independent sublibraries, each of which may be "distributed" according to sense (1) above. (3) An affiliation of logically distinct

asset libraries to accomplish some common purpose. The term "distributed" often carries the connotation that the various libraries and databases are geographically separated, sometimes by global distances.

HETEROGENEOUS ASSET LIBRARY A library with dissimilarities among its components. Especially, a distributed library having dissimilarities in computer platforms, operating systems, database or object management systems, and library data models and mechanisms.

DISTRIBUTED, HETEROGENEOUS ASSET LIBRARY An asset library that is implemented across distributed, heterogeneous computer platforms and contains heterogeneous asset data models.

FILE SYSTEM ASSET LIBRARY An asset library whose implementation is based on file system facilities providing management of assets as files in a directory hierarchy with file name searching, intra-file pattern matching searching, and access control based on the file system facilities.

FRAMEWORK ASSET LIBRARY An asset library system whose search and understanding support is implemented using a software environment integration framework.

ASSET LIBRARY MECHANISM A software (sub)system that provides a logical capability for a library. A library mechanism requires tailoring and, possibly, extension to become a library system instantiation.

STANDALONE ASSET LIBRARY An asset library system whose search and understanding support is implemented independently of a particular file system capability or software environment integration framework.

TOOL (1) An independent software program that uses ALOAF services to accomplish some specific task. (2) A program which provides the functionality of one or more ALOAF services at the request of an ALOAF server or binding routine.

COMPONENT One of the parts that make up a software-intensive system. A component may be hardware or software and may be subdivided into other components. A complete software component includes both the object code and all related information that is needed to use it. This related information includes parameterization information, source code if not proprietary, test information, design information, evaluation results, and other descriptive information.

DATA MODEL The organizing principles and concepts underlying structured data, as in a database. Also, the means of representing that structure.

DOMAIN An area of activity or knowledge. Domains have been characterized as application, horizontal or vertical, technology, computer science, execution, execution models, etc.

DOMAIN ANALYSIS The process of identifying, collecting, organizing, analyzing, and representing a domain model and software architecture from the study of existing systems, underlying theory, emerging technology, and development histories within the domain of interest.

DOMAIN ENGINEERING The construction of components, methods, and tools and their supporting documentation to solve the problems of system/subsystem development by the application of the knowledge in the domain model and software architectures.

DOMAIN MODEL A definition of the functions, objects, data, requirements, relationships and variations in a particular domain.

DOMAIN-SPECIFIC LANGUAGE A machine processable language whose terms are derived from the domain model and that is used for the definition of components or software architectures supporting that domain.

FRAMEWORK A skeletal structure to support or enclose something. The skeletal structure in ALOAF reuse documents is a conceptual structure that delimits the concepts being discussed; supports understanding and technical transition; and promotes evolution.

REUSE PROCESS FRAMEWORK The conceptual structure that categorizes and interrelates reuse processes by their purposes, goals, and activity characterizations.

ASSET LIBRARY OPEN ARCHITECTURE FRAMEWORK The conceptual structure that supports seamless interchange and interoperability among networked, distributed, heterogeneous asset libraries by defining a service model; protocols supporting that model; Ada package specifications for the protocols; and a specification for asset interchange.

INTEROPERABILITY The capability to perform common functions or processes across the boundaries created by the connections between homogeneous and heterogeneous asset libraries' host computers.

LIBRARY (see ASSET LIBRARY)

LIFE CYCLE All the activities a software or software-related product is subjected to from its inception until it is no longer useful. Note that this definition shifts the usual definition of life cycle, which is based on life of a *system, to a more general* concept covering the lifetime of a software *product.*

PLUG-COMPATIBLE Usable in combination without modification.

META-MODEL A modeling technique used to develop a class of data models. A data model is said to conform to a meta-model if it can be completely described by that meta-model. (Also called META-DATA-MODEL.)

PORTABILITY The quality of a software product, tool, component, etc. that determines the amount of manual effort needed to adapt it to a new operating environment. In particular, the ability to move ALOAF tools and service frameworks from one asset library system to another with no effort other than recompilation and relinking.

PROCESS A series of steps, actions, or activities to bring about a desired result.

SOFTWARE DEVELOPMENT PROCESS A process whose goal is the development of software components or applications.

QUERY A request for identification of a set of assets or library data model elements, expressed in terms of a set of criteria that the identified items must satisfy.

REUSE The transfer of expertise. In software engineering, reuse often refers to the transfer of expertise encoded in software related work products. The simplest form of reuse from software work products is the use of subroutine/subprogram libraries for string manipulations or mathematic calculations. The simplest form of reuse of expertise not represented in software work products is the employment of a human experienced in the desired endeavor.

REUSE-BASED DEVELOPMENT The application of a disciplined, systematic, quantifiable approach to the development, operation and maintenance of software with reuse as a primary consideration in the approach.

SEAMLESS An operational mode where a user is able to perform reuse processes across any boundaries of interoperating reuse systems without noticing any of the extra work done by the asset library to maintain the communications between the heterogeneous libraries. Use of a seamless interoperating asset library should be indistinguishable from the use of a local asset library to which a user is directly connected.

SERVICES The functionality collectively provided by an asset library's framework that provide reuse-oriented tools with the features they need.

SERVICE MODEL The categories and inter-relationships among reuse services that provide an organization (or at least the appearance of an organization) to other tool and software developers.

SERVICE PROTOCOLS The syntax and semantics of how each service and the data it provides and/or uses are employed by other services or tools.

SERVICE CATEGORIES A grouping of reuse services by common functionality, purpose, and/or protocols.

PROGRAMMATIC INTERFACE A binding of a service protocol to the specific syntax and semantics provided by a computer software programming language, such as Ada.

SOFTWARE ARCHITECTURE The high level design for a software system or subsystem. Includes the description of each software component's functionality (or result), name, parameters and their types and a description of the components' interrelationships. Note that this definition describes software architecture from a system point of view rather than a domain point of view. Many different definitions of software architecture are currently in use, often in the same sentence depending upon qualifiers such as "generic" or "domain-specific."

SOFTWARE ENGINEERING ENVIRONMENT (SEE) The computer hardware, operating system, tools, computer-hosted capabilities, and rules that an individual software engineer works within to develop a software system.

SOFTWARE ENGINEERING ENVIRONMENT FRAMEWORK A set of capabilities that integrate user interface, project data, network communications, and control of resources in a coherent manner.

STANDARDS Acknowledged measures of comparison for quantitative or qualitative value. ALOAF addresses itself to: formal standards, which have been fully adopted by appropriate accredited national and/or international standards organizations such as ANSI and ISO; in-work standards activities, which represent work by accredited and other organizations aimed at creation or updating of formal standards (such as CDIF and IEEE P1175); and proposal of new standards in areas not fully covered by formal and in-work standards.

B Acronyms

- **ACL** – Access Control List
- **ADT** – Abstract Data Type
- **ALOAF** – Asset Library Open Architecture Framework
- **AMS** – Asset Management System
- **ANSI** – American National Standards Institute
- **ASCII** – American Standard Code for Information Interchange
- **ASN.1** – Abstract Syntax Notation One
- **ATIS** – A Tools Integration Standard
- **BNF** – Backus-Naur Form
- **CALS** – Computer-aided Acquisition and Logistics Support
- **CASE** – Computer-Aided Software Engineering
- **CDIF** – CASE Data Interchange Format
- **CDM** – Common Data Model
- **CM** – Configuration Management
- **CMS** – Code Management System
- **CONOPS** – Concept of Operation
- **COTS** – Commercial Off-The-Shelf
- **CPU** – Central Processing Unit
- **DBMS** – Data Base Management System
- **DoD** – Department of Defense
- **ECMA** – European Computer Manufacturing Association
- **EIA** – Electronic Industries Association
- **EIS** – Engineering Information System
- **ERA** – Entity Relationship Attribute
- **IEEE** – Institute of Electrical and Electronic Engineers
- **IRDS** – Information Resource Dictionary System
- **ISO** – International Standards Organization
- **ISEE** – Integrated Software Engineering Environment

- **IV&V** – Independent Validation and Verification
- **OODBMS** – Object Oriented Data Base Management System
- **OMS** – Object Management System
- **OS** – Operating System
- **OSF** – Open Software Foundation
- **PDL** – Program Design Language
- **PDSS** – Post Deployment Support System
- **PCTE** – Portable Common Tool Environment
- **POSIX** – Portable Operating System Interface for computer environments
- **RCS** – Revision Control System
- **RIG** – Reuse (library) Interoperability Group
- **RDBMS** – Relational Data Base Management System
- **RLF** – Reusability Library Framework
- **RM** – Reference Model
- **SCCS** – Source Code Control System
- **SEE** – Software Engineering Environment
- **SGML** – Standard Generalized Mark-up Language
- **SQL** – Structured Query Language
- **SSP** – STARS Standards Portfolio
- **STL** – Semantic Transfer Language
- **STARS** – Software Technology for Adaptable, Reliable Systems
- **STEP** – Standard for the Exchange of Product (Model Data)
- **TBD** – To Be Determined
- **UIMS** – User Interface Management System
- **VDM** – Vienna Design Method

C Scenarios

This appendix will contain scenarios of reuse activities that show, through various views, how the ALOAF services interact to support the activities.

D Asset Interchange Language Specification

This section describes the asset interchange language which will be used to support the short term approach to asset interchange. This language is designed to represent the Common Data Model and does not permit direct expression of the data model of the exporting library. Neither is this language based on any existing data interchange standards. It is a simple, ad-hoc language to provide a basic asset interchange capability.

BNF of the Asset Interchange Language

```
object_list ::= "begin" "object_list" { object } "end" "object_list"

object ::= asset | file | organization | person

object_attribute ::= unique_identifier

asset ::= "begin" "asset" { asset_attribute } "end" "asset"

asset_attribute ::= object_attribute |
                    name |
                    alternate_name |
                    version |
                    release_date |
                    description |
                    restrictions_apply |
                    is_comprised_of |
                    is_ancestor_of |
                    is_descendant_of |
                    requires |
                    is_required_by |
                    was_created_by |
                    is_understood_by

file ::= "begin" "file" { file_attribute } "end" "file"

file_attribute ::= object_attribute |
                   file_name |
                   comprises

organization ::= "begin" "organization" { organization_attribute } "end"
  "organization"

organization_attribute ::= object_attribute |
                           name |
                           alternate_name |
                           address |
                              telephone_number |
                              created

person ::= "begin" "person" { person_attribute } "end" "person"

person_attribute ::= object_attribute |
                     name |
                     address |
                     telephone_number |
                     electronic_mail_address |
                     is_contact_for

address ::= "address" "=>" string_literal

alternate_name ::= "alternate_name" "=>" string_literal

comprises ::= "comprises" "=>" identifier

created ::= "created" "=>" identifier

description ::= "description" "=>" string_literal

electronic_mail_address ::= "electronic_mail_address" "=>" string_literal

file_name ::= "file_name" "=>" string_literal

is_ancestor_of ::= "is_ancestor_of" "=>" identifier

is_comprised_of ::= "is_comprised_of" "=>" identifier

is_contact_for ::= "is_contact_for" "=>" identifier

is_descendant_of ::= "is_descendant_of" "=>" identifier

is_required_by ::= "is_required_by" "=>" identifier

is_understood_by ::= "is_understood_by" "=>" identifier

name ::= "name" "=>" string_literal

release_date ::= "release_date" "=>" string_literal

requires ::= "requires" "=>" identifier

restrictions_apply ::= "restrictions_apply" "=>" string_literal

telephone_number ::= "telephone_number" "=>" string_literal
```

```
unique_identifier ::= "unique_identifier" "=>" identifier

version ::= "version" "=>" string_literal

was_created_by ::= "was_created_by" "=>" identifier
```

BNF Design Notes

a. The symbol `string_literal` differs from the usual Ada definition in that a string literal may include format effectors such as a line feed or a carriage return. The symbol `identifier` has the usual Ada definition.

b. The idea of using a modified form of the Ada language syntax for aggregates was considered and rejected. The advantage of using the Ada aggregate syntax is that it is well-defined and understood by the STARS audience. On the down side, the aggregate syntax is overkill for the purpose at hand. The syntax given above is simple enough to be self-explanatory, and satisfies our immediate needs.

c. We need to extend the syntax to include certain "header" information not specified by the Common Data Model. For example: the date of export, an identification of the exporting library (and its environment?), and perhaps the name of the person who initiated the export.

d. For our purposes, there was no need to instantiate the `object` class.

e. Since each object is clearly introduced with a `begin class-name` clause, there is no need to include a `class_name` attribute.

f. Since the symbol `identifier` is used to represent an Ada identifier, the `identifier` attribute of the `object` class is referred to by the symbol `unique_identifier`.

g. Certain rules that can be enforced in the syntax of the language have been postponed to the semantic level. In particular, the syntax definition given above says nothing about whether an attribute is mandatory or optional, or whether a given attribute can be repeated within a single object. These rules could have been enforced in the BNF by requiring that the attributes for each object class be given in a certain order, with each attribute specified as mandatory or optional. Because such a syntax would be unnecessarily clumsy, these rules should be enforced at a higher level.

E Ada Bindings for ALOAF Services

This appendix will contain the Ada bindings to the language independent specifications for the ALOAF services defined in section 8.